Learn by doing: less theory, more results

Google App Inventor

Create powerful Android apps the easy all-visual way with Google App Inventor

Beginner's Guide

Ralph Roberts

[PACKT] PUBLISHING

Google App Inventor

Beginner's Guide

Download from Wow! eBook <www.wowebook.com>

Create powerful Android apps the easy all-visual way with Google App Inventor

Ralph Roberts

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Google App Inventor

Beginner's Guide

Copyright © 2011 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2011

Production Reference: 1171011

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-212-0

www.packtpub.com

Cover Image by Fillippo (filosarti@tiscali.it)

Credits

Author

Ralph Roberts

Reviewers

M. Hossein Amerkashi

Tony Barnes

Roger Belk

Bart Van Den Brande

J.S. McClellan

Acquisition Editor

Rashmi Phadnis

Development Editor

Hyacintha D'Souza

Technical Editors

Arun Nadar

Ajay Shanker

Project Coordinator

Srimoyee Ghoshal

Proofreader

Jonathan Todd

Indexer

Tejal Daruwale

Monica Ajmera Mehta

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Author

Ralph Roberts is a decorated Vietnam Veteran and worked with NASA during the Apollo moon program. He built his first personal computer in 1976 and has been writing about them and on them since his first published article, "Down with Typewriters", in 1978. He has written over 100 books along with thousands of articles and short stories. His best sellers include the first U.S. book on computer viruses (which resulted in several appearances on national TV) and *Classic Cooking with Coca-Cola®*, a cookbook that has been in continuous print for the past 17 years and sold half a million copies. He is also a video producer with over 100 DVD titles now for sale nationally on places such as `Amazon.com`. He has also produced hundreds of hours of video for local TV in the Western North Carolina area and sold scripts to Hollywood producers. And, he has a number of apps on Android Market written with App Inventor—search for ArrSoft (his software company) to see them. Previously for Packt Publishing, Ralph wrote *Celtx: Open Source Screenwriting*. Ralph and his wife Pat live on a farm in the mountains of Western North Carolina.

About the Reviewers

M. Hossein Amerkashi is a Java/J2EE Software Engineer/consultant in Maryland.

Hossein, a full time Software Engineer/consultant, has been in the IT field for over 15 years. He is also the CEO of 3N Portal LLC, an IT consulting company, and works on large-scale Enterprise Application Software (EAS) development. He has extensive experience in Android Software Development using Google App Inventor, Android SDK, and App Inventor JavaBridge. He has a prominent role in the App Inventor community. He lives in Maryland with his wife and three kids.

Hossein has developed AppToMarket tool that has helped thousands of AI developers to auto-convert and publish their AI apps to Android Market. He has also developed and published many tutorials in his Weblog, apptomarket forum, and code repository.

I would like to thank my family, especially my wife Lila, who have been very patient with me for putting so many hours into developing tools and tutorials to help the AI community.

Tony Barnes has been an IT Consultant for over 20 years. He is the founder of tAIR.info and has been working with App Inventor since 2010. He is also a forum moderator for the Google App Inventor Coffee Shop and Programming forums. He likes helping people learn App Inventor.

tAIR – The App Inventor Repository-was started by him as a community-based site. Along with his right-hand helpers Gary Frederick and Gene Kupfer, he built a helpful community site and they are all very active within the App Inventor community. They host a weekly App Inventor Enthusiast webchat at 5pm Eastern U.S. time. You can get to it by pointing your browser to `www.tair.info/webchat`.

Roger Belk, aka Big Daddy App, has been developing Android apps for the past year. He is a self-taught 43 year-old Ironworker. He builds apps using Eclipse with the Android SDK, Java, and Google's App Inventor. You can check out his website at `www.BigDaddyApp.com`.

Roger is also a power user in Google's App Inventor forums, answering help requests from new AI developers, from the set up, How-To, and the Coffee Shop, just chatting and kicking around ideal for AI apps.

He has also worked on Animation 3.0 and Google App Inventor books.

J.S. McClellan has been working with computers for the past 25 years in various forms/ fashions and is a computer/technology enthusiast. From programming with Apple Basic, Pascal, C+, to currently working with App Inventor and learning Eclipse, McClellan also enjoys learning and experiencing new things/technologies including programming languages that make it easier for people to make things which they can use. That is what drew McClellan's attention to Google's App Inventor. The ease-of-use of the product makes it enjoyable to use and help others to use too.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface 1

Chapter 1: Obtaining and Installing Google App Inventor 7

Getting a Google App Inventor Account 8
Time for action – signing up for a Google Account 8
Google App Inventor on the Web 9
Time for action – logging into App Inventor 9
Requirements for Windows, Mac, and Linux 13
Obtaining and installing Java 14
Installing Java on your computer 15
Installing App Inventor locally 15
Time for action – installing App Inventor on Mac 16
Time for action – installing App Inventor on GNU/Linux 16
Time for action – installing App Inventor on Windows 18
Setting up and running the emulator 19
Time for action – opening up the emulator 20
Finding and downloading drivers 25
Mac 26
Linux 26
Time for action – connecting a Droid to Ubuntu 26
Windows 28
Time for action – configuring our device 29
Summary 33

Chapter 2: Learning Components 35

Using App Inventor's web interface 36
My Projects 37
Time for action – creating a new project 38
Time for action – downloading our projects 40

Time for action – uploading source code **42**
Design 44
Time for action – connecting our phone **48**
Designing our apps 50
Basic components **51**
Button 52
Time for action – adding and configuring a button **53**
Canvas 58
Time for action – follow the bouncing ball **58**
Checkbox 59
Clock 60
Image 60
Label 60
ListPicker 61
PasswordTextBox 62
TextBox 62
TinyDB 63
Media components **64**
Camera 65
Time for action – shooting a photo **65**
ImagePicker 67
Player 67
Sound 69
Time for action – vibrating buttons **69**
VideoPlayer 70
Animation components **70**
Ball 71
ImageSprite 71
Social components **72**
ContactPicker 72
EmailPicker 72
PhoneCall 73
PhoneNumberPicker 73
Texting 73
Twitter 74
Sensor components **74**
Screen arrangement components **74**
LEGO® MINDSTORMS® components **75**
Other stuff **75**
Not ready for prime time **75**
Summary **75**

Download from Wow! eBook <www.wowebook.com>

Chapter 3: Playing with Blocks 77
The Blocks Editor 78
The top bar 78
Time for action – collapse and expand block groups 82
Time for action – deleting unwanted blocks 84
Definition blocks 85
procedureWithResult 86
Time for action – calculating the area of a circle using a function Design 88
Time for action – calculating the area of a circle using a function 92
Time for action – finishing the "pi are square" app 94
Time for action – adding an error trap 97
Procedure 98
Variable 100
Name 101
Dummy 103
Text Blocks 104
List blocks 112
CSV list blocks 120
Math blocks 122
Logic blocks 126
Control blocks 127
Time for action – making a loop 128
Colors blocks 129
Individual component blocks 131
Summary 132
Chapter 4: Mastering Concepts and Advanced Components 133
Changelog 134
Concepts 135
Live development, testing, and debugging 135
Specifying sizes of components 136
Accessing images and sounds 137
Other stuff 138
ActivityStarter 138
Time for action – building bookmark apps 139
BarcodeScanner 143
Time for action – our very own barcode scanner 143
Bluetooth 144
Notifier 148
SpeechRecognizer 154

Time for action – build a time- and date-stamped Voice Note Taker **154**
TextToSpeech 156
Time for action – reading aloud **157**
TinyWebDB 159
Time for action – storing persistent data on the web **159**
Web 161
Not Ready for Prime Time **163**
FusiontablesControl 163
GameClient 165
SoundRecorder 165
Voting 165
What we learned **166**

Chapter 5: Apps That Communicate **167**

App—phone home **167**
CallFriends app 171
Time for action – getting the source **172**
Texting with words and more **178**
Time for action – building a texting app **178**
Time for action – create a text-your-friends app **182**
Applications e-mailing stuff **184**
Time for action – that Loving feeling **186**
Social communication **193**
Time for action – two methods for searching Twitter **193**
Facebook and other social media sites 196
A bonus template **197**
Summary **198**

Chapter 6: Apps That Remember **199**

Lists and Lists of Lists **200**
Static lists 200
Changeable lists 202
Time for action – building the input screen for a Friends List **202**
Handling database records 207
Time for action – converting a CSV table **209**
TinyDB—persistent to please **211**
Web component—reaching out to the web **212**
Parsing 213
Time for action – using parsing to break out the states of India **214**
Get 217
Summary **219**

Chapter 7: Apps That Surf the Web 221
Browsing and using websites 221
Time for action – building an eBay link app 222
Fusion Tables—Google's free online data service 227
Quickie data-getter—finding the fourth way 236
TinyWebDB—accessing and storing data 237
Time for action – testing TinyWebDB 238
Publishing your apps for the world 243
Summary 247
Chapter 8: Apps That Know Where They Are 249
Changelog 249
Bringing web pages into our apps 250
Time for action – showing three websites at the same time inside an app 250
Using the Post method with the Web component 252
Advanced tab in Blocks Editor 254
Other changes 256
Son of Changelog 256
Using the location sensor 257
Using Google Maps 258
Time for action – determining our location by network 258
Time for action – determining our location by GPS 259
Time for action – taking a look from space 260
Using the AccelerometerSensor component 263
Time for action – seeing gravity and acceleration on our phones 264
Time for action – nice and level 265
Which way are we? 269
Time for action – seeing azimuth, pitch, and roll 269
Time for action – building a compass 270
How high are we? 272
Time for action – finding your current altitude 272
How far from home are we? 273
Time for action – creating the end and the beginning 275
Time for action – getting the current address, longitude, and latitude 276
Time for action – saving our home location 277
Time for action – the distance between home and wherever here is 279
Summary 281

Chapter 9: Games and Animation! 283
Animation 283
Time for action – bouncing a ball 284
Time for action – banging things off each other 289
Some more basics of App Inventor animation 295
Splash screens 297
Time for action – designing a splash screen 297
Time for action – making a splash with the splash page 300
Bash the Alien! 303
Time for action – throwing blocks to Bash the Alien! 305
Knowledge games 307
GameClient component 314
Summary 315
Thank you! 315
Appendix A: Links and Resources 317
Websites 317
Troubleshooting 318
Using the Android SDK 320
Java Bridge 321
Appendix B: Last-Minute Update 323
Status of App Inventor 324
Last-minute update 324
Appendix C: Final Last-Minute Update 327
Letter from the App Inventor Team 327
Appendix D: Pop Quiz Answers 329
Chapter 1 329
Obtaining and Installing Google App Inventor 329
Chapter 3 329
Playing with Blocks 329
Chapter 5 330
Apps That Communicate 330
Chapter 7 330
Apps That Surf the Web 330
Index 331

Preface

Google App Inventor is from the Internet giant Google, home of the famous search engine. Google also develops the Android operating system found on smartphones and tablet computers from many manufacturers including HTC, Motorola, Samsung, and a growing host of others.

In late breaking news as this book goes to press, Google and MIT (Massachusetts Institute of Technology) have announced that App Inventor will be supported by MIT's new Center for Mobile Learning. People from MIT (such as Hal Abelson and Mitchel Resnick) who first developed App Inventor will continue to be associated with its development. More about this change at the end of this book.

Some reviews of App Inventor call it limited. This is wrong. Google App Inventor is already powerful and ready to produce complex useful and marketable apps (as I and many others have already accomplished using it). Additionally, the software's still in beta (development and testing stage) — new features and capabilities continue to be added. But, don't just take my word for how useful App Inventor is; let me show you throughout this book.

Until App Inventor was introduced, programming apps (applications) was an esoteric art form involving cryptic lines of Java code understood only by a few programmers. You were required to write line upon line of cryptic code. Now, we can just pull blocks together in easy visual programming. Can anyone do it? Yes, we can!

What this book covers

Chapter 1, *Obtaining and Installing Google App Inventor*: here, we create a personal account on the Google App Inventor site, set up Java on our computer, and connect our phone or other Android device to our computer.

Chapter 2, *Learning Components*: here, we choose and use various components. Components may be thought of as services. For example, basic components include buttons, labels, checkboxes, and so forth. Media components allow us to play sounds, show videos, and so forth. Social components let us interact with others by phone calls, e-mail, texting, Twitter, and so on.

Chapter 3, *Playing with Blocks*: here, Blocks (of code, but we never have to see the code) let us tell the components what we want them to accomplish. It's all visual now, dragging-and-dropping blocks into various configurations. Blocks are simply the logic that instructs the component services how to act and in what order tasks happen. In this chapter, we look at each type of block and how it is used and create some useful apps.

Chapter 4, *Mastering Concepts and Advanced Components*: here, Concepts extend the power of our apps, giving us access to even more than can be accomplished with components and blocks. The Activity Starter lets one app call another. A TinyWebDB lets us set up a small database on the web so that apps can access data as needed rather than storing it all in the limited space on a smartphone or other Android device. The location sensor gives us access to GPS, letting us write apps such as those that would track our travel, tell us our speed and elevation, and more. In this chapter, we explore concepts and master their use, building several neat example apps.

Chapter 5, *Apps That Communicate*: here, Android devices right now are primarily phones. But, even tablet computers (pads) need to communicate with the world. This chapter covers apps that call, text, e-mail, and even Tweet.

Chapter 6, *Apps That Remember*: here, any Android device is really a small (yet powerful) computer. Computers excel in collecting and manipulating data. In this chapter, we devise apps that remember data for us, help us collect it, and present it to us by whatever criteria we specify. The apps we pull together in this chapter will remember stuff (accumulate and return data).

Chapter 7, *Apps That Surf the Web*: mostly, every Android device connects to the Internet. Our apps can also! This chapter shows us how to use networks and the Internet to exchange data and more. With the power of apps, you'll find an unlimited data plan pays for itself soon enough. Let's build some apps that are web-aware!

Chapter 8, *Apps That Know Where They Are*: here, we come to know that Android devices have a lot of built-in sensors and features that use them. Many of those we've already used, but what about location awareness? Your phone knows where it is, or can find out quickly. And, we can build apps using that power and much more relating to maps, navigation, and location!

Chapter 9, *Games and Animation!*: historically, creating video games has been a long and complex undertaking. All that has now changed! Google App Inventor has all we need to construct some very powerful and entertaining games with very little effort. In this chapter, we unleash our imaginations! Prepare to have fun—before and after dragging blocks into some neat games.

Appendix A, *Links and Resources*: mentions some useful websites and additional software, which all augment the power of App Inventor.

Appendix B, *Last-Minute Update*: Like any good open source software with a growing user base, changes occur. Google is releasing App Inventor to world-famous MIT (Massachusetts Institute of Technology), which created it in the first place. This is a strong move for AI—which can only improve it—and here are some details as I know them so far.

Appendix C, *Final Last-Minute Update*: here, are the last-minute updates on this process and that App Inventor is changing from Google back to MIT, where it was first created. This is a very positive change for the growth of AI.

What you need for this book

App Inventor basics—both in hardware and prior knowledge—are minimal. You need only a computer and an Internet connection. It helps to have a smartphone or other Android device, but not necessary as you can emulate (use software on your computer, which we will install in *Chapter 1*, *Obtaining and Installing Google App Inventor* to test apps).

As to prior knowledge and programming experience needed in creating apps and publishing them to places such as the Android market, you need practically none! This is the truly beautiful part of AI: its visual interface allows anyone from elementary school kids to us older people to make apps from the beginning without special knowledge.

Who this book is for

Again, because of the easy visual interface in which blocks are dragged together to create the operational logic of apps, just about all of us can quickly master and build working, useful apps.

In short, this book is for all of us, and especially for those who have never programmed before. AI makes it simple.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Open up a new project or the `my_spiffy_new_app`, the one we created earlier".

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on **More information** in the mini dialog box above for additional info".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt Publishing, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Download from Wow! eBook <www.wowebook.com>

1

Obtaining and Installing Google App Inventor

In this chapter, we create a personal account on the Google App Inventor site, set up Java on our computer (which runs the Blocks Editor), and connect our phone or other Android device to our computer (so we can test our apps). We then create our first app and see it work on our device.

What we learn in this chapter:

- Signing up for the free Google App Inventor account. Part of App Inventor runs on the web (the Designer) and part of it on our local computers (the Blocks Editor).
- Logging onto the App Inventor website.
- Requirements for PC, Mac, and Linux.
- Obtaining and installing Java.
- Downloading and installing the part of App Inventor that runs locally.
- Running the Emulator (a cute but fake virtual smartphone on your local computer for testing apps).
- Finding and downloading device drivers (for our phone or other Android device).
- Configuring our device to work with App Inventor.

Once we set up our working environment—that is, we have an App Inventor account, Java and the local part of Google App Inventor on our local machine, and either an emulator (virtual Android device) or our phone connected via USB cable—we are ready to begin.

The good news is that the only thing approaching any degree of difficulty in using App Inventor is this initial installation process. Do that once, and it is forever out of the way.

The rest of the good news is that the bad news was cancelled as soon as we heard it was possible to create powerful apps for Android devices without ever writing a line of Java code!

Drag-and-drop—the new way of programming. But, let's hold the party until we get our working environment…er… working.

Getting a Google App Inventor Account

If you already have a Google account—which gives you access to lots of Google web applications such as Gmail, YouTube, and many more—you already have an App Inventor account. Just sign in (see the next section for how) and use it.

For most of last year (2010), Google App Inventor was a closed beta testing program—meaning one had to be invited to participate. I was and did, which gave me a small headstart in learning App Inventor, but I'll be helping all you guys catch up and surpass me.

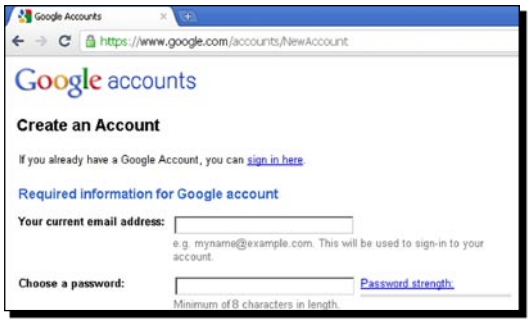
In December 2010, Google opened the App Inventor beta program to everyone holding a Google account. App Inventor is still in beta but that means little, since Goggle tends to keep applications in "testing" for longer than many software publishers.

The point here is that App Inventor is now open to everyone and you, I, and Aunt Mabel can start writing and publishing our own apps.

Time for action – signing up for a Google Account

If you do not have a Google account yet, it's simple enough to get one. Free, of course.

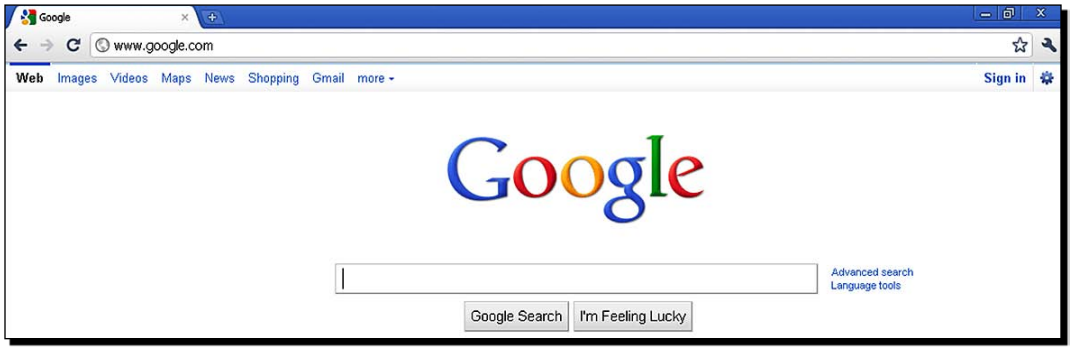
1. Browse to `https://www.google.com/accounts/NewAccount` (see the following sign-up page).
2. Create your account, and you are good to go. Again, it's free.


Google Accounts
https://www.google.com/accounts/NewAccount
Google accounts
Create an Account
If you already have a Google Account, you can sign in here.
Required information for Google account
Your current email address:
e.g. myname@example.com. This will be used to sign-in to your account.
Choose a password:
Password strength:
Minimum of 8 characters in length.

What just happened?

With a Google account, you can use Google applications on the web and, most important to us right now, log into the App Inventor site on the web.

Once you have a Google account, you can sign in anytime by going to `http://google.com` and clicking on **Sign in** (upper-right of the main screen as shown in the following screenshot):



Google App Inventor on the Web

Once you have a valid Google account, the App Inventor site with tutorials, the forum, links to additional information, and much more is now open for use.

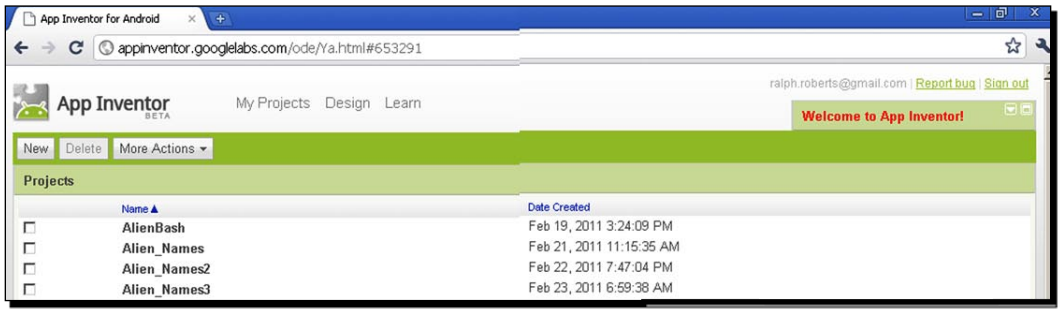
Under that "much more" term falls the App Inventor **Designer**, the web portion of this visual app-designing system (the rest being the Java application **Blocks Editor** that downloads to your local computer, and a testing device such as your own smartphone or emulator software on your computer to generate a virtual device).

Time for action – logging into App Inventor

To sign on to App Inventor, go to `http://appinventor.googlelabs.com`.

This URL will be changing sometime soon, but a redirect should be available. If asked, log in with your Google account information again—once logged in, you'll be remembered, and logging in should be automatic. The first time, you'll also be asked to agree to the Terms of use).

Once logged in, you will see something like the following screenshot. This is mine, so it has some ongoing projects listed; your first-time entry will, of course, have no projects listed yet.



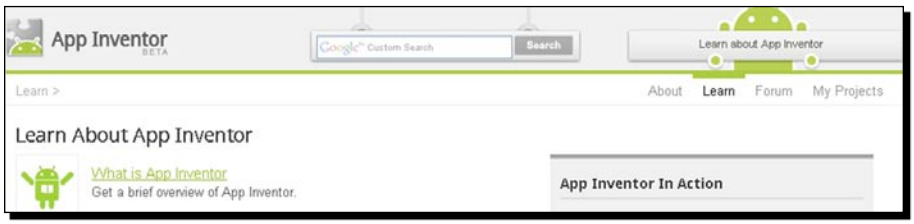
What just happened?

We have now entered the area where apps are designed.

This area contains the App Inventor Designer, which allows us to choose and place components (items such as buttons, sound players, labels, and so on. *Chapter 2*, *Learning Components* covers components in detail). It also stores projects for us (four of mine may be seen listed previously).

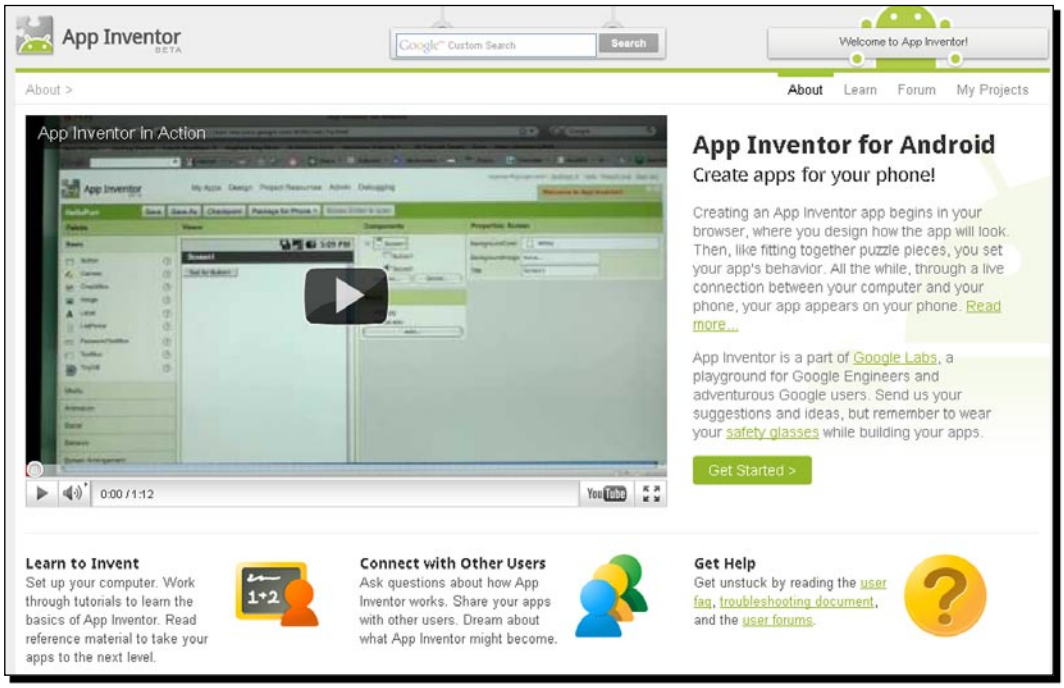
Left-clicking on a project's title opens it up for editing in the **Designer**. Or, a new project may be started by clicking on the **New** button. Clicking on **Design** just to the right of the App Inventor Beta logo opens the last project we were working on.

Left-clicking on **Learn** opens up the informational part of the App Inventor site (see the following illustration):



Let's look briefly at the four buttons on the upper-right under the little Droid holding the **Learn about App Inventor** sign. We'll click on each of these in turn and introduce the area they bring up.

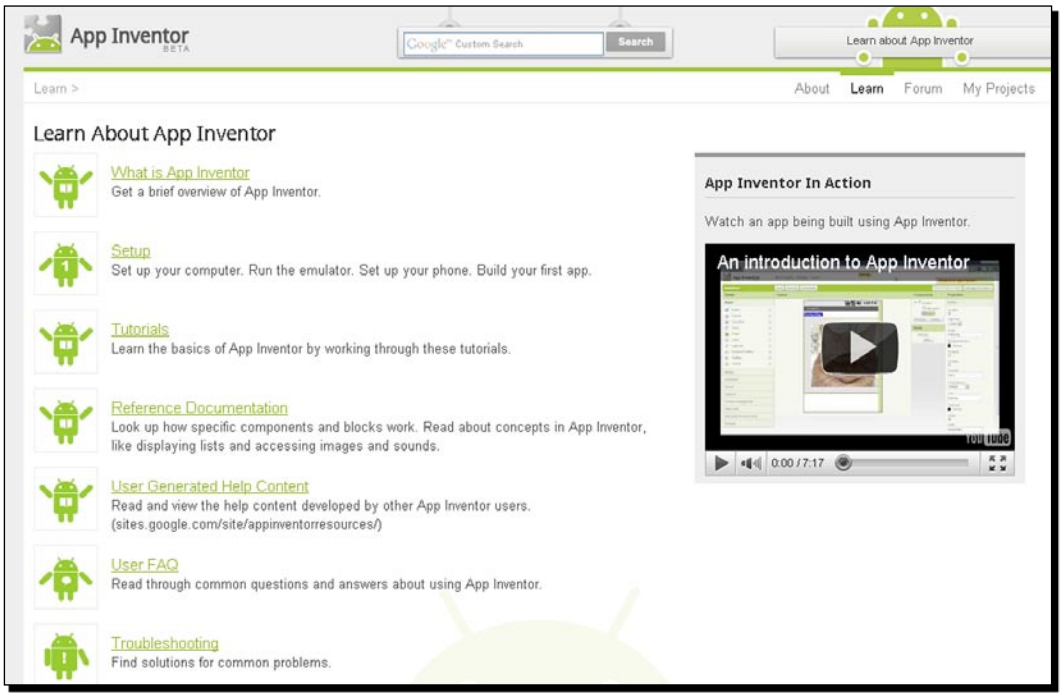
About: Clicking on this button takes us to introductory material and links to more information (see the lower part of the following screenshot) such as **Learn to Invent**, **Connect with Other Users**, and **Get Help**.


App Inventor
Search
Welcome to App Inventor!
About >
About
Learn
Forum
My Projects
App Inventor in Action
App Inventor for Android
Create apps for your phone!
Creating an App Inventor app begins in your browser, where you design how the app will look. Then, like fitting together puzzle pieces, you set your app's behavior. All the while, through a live connection between your computer and your phone, your app appears on your phone. Read more...
App Inventor is a part of Google Labs, a playground for Google Engineers and adventurous Google users. Send us your suggestions and ideas, but remember to wear your safety glasses while building your apps.
Get Started >
0:00 / 1:12
Learn to Invent
Set up your computer. Work through tutorials to learn the basics of App Inventor. Read reference material to take your apps to the next level.
Connect with Other Users
Ask questions about how App Inventor works. Share your apps with other users. Dream about what App Inventor might become.
Get Help
Get unstuck by reading the user faq, troubleshooting document, and the user forums.

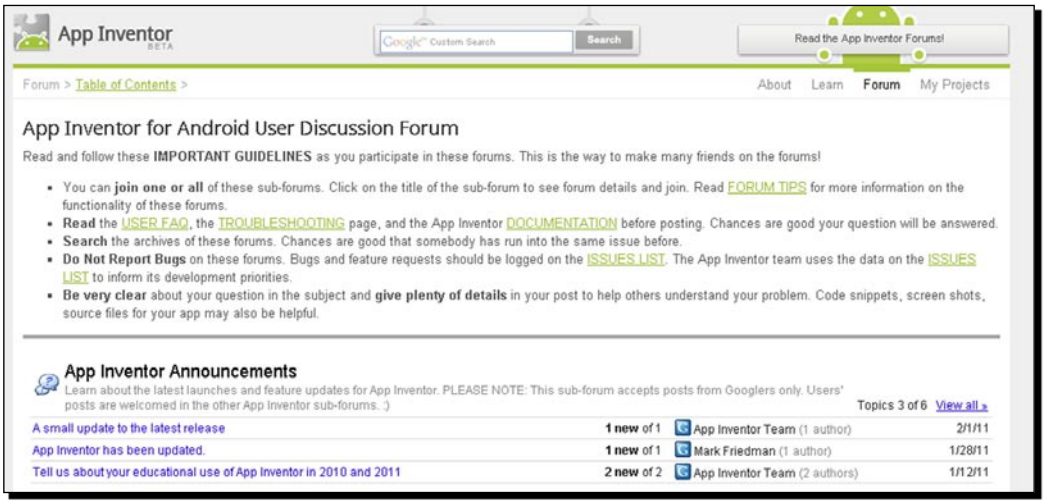
The **App Inventor in Action** video is a quick watch that gives a feel for how App Inventor works.

Learn: Clicking on **Learn** gives us links (shown in the next screenshot) to tutorials, setup information, reference documentation, user-generated help and tutorials (this section is gold-lots of good stuff and always growing), FAQs (Frequently Asked Questions), and troubleshooting hints (gee, not even all of my apps work the first time).

The video on the following screenshot—**An introduction to App Inventor**—is just over seven minutes long and a nice little intro to how App Inventor works.

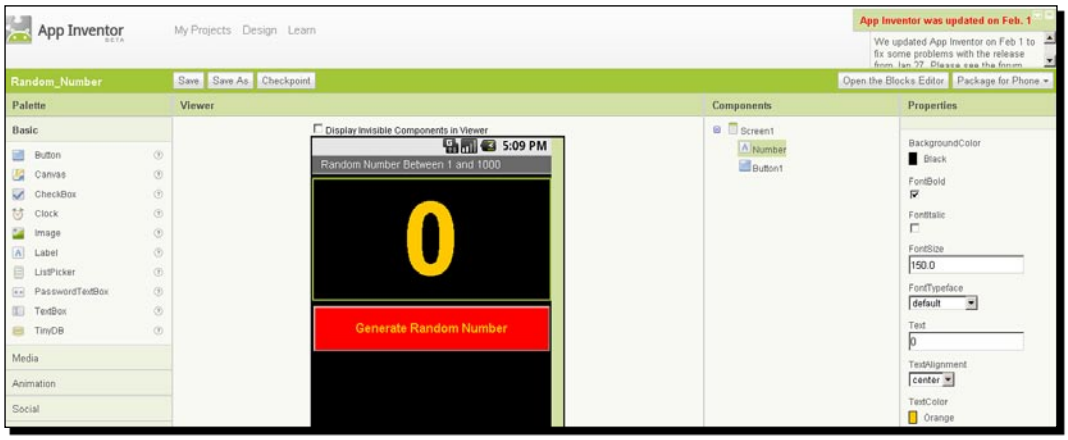

App Inventor
BETA
Search
Learn about App Inventor
Learn >
About
Learn
Forum
My Projects
Learn About App Inventor
What is App Inventor
Get a brief overview of App Inventor.
Setup
Set up your computer. Run the emulator. Set up your phone. Build your first app.
Tutorials
Learn the basics of App Inventor by working through these tutorials.
Reference Documentation
Look up how specific components and blocks work. Read about concepts in App Inventor, like displaying lists and accessing images and sounds.
User Generated Help Content
Read and view the help content developed by other App Inventor users. (sites.google.com/site/appinventorresources/)
User FAQ
Read through common questions and answers about using App Inventor.
Troubleshooting
Find solutions for common problems.
App Inventor In Action
Watch an app being built using App Inventor.
An introduction to App Inventor
0:00 / 7:17

Forum: The **Forum** shown in the next screenshot allows us to interact with other users of Google App Inventor as well as some of the Google development staff. A place to both ask and (as our own knowledge grows) answer questions.

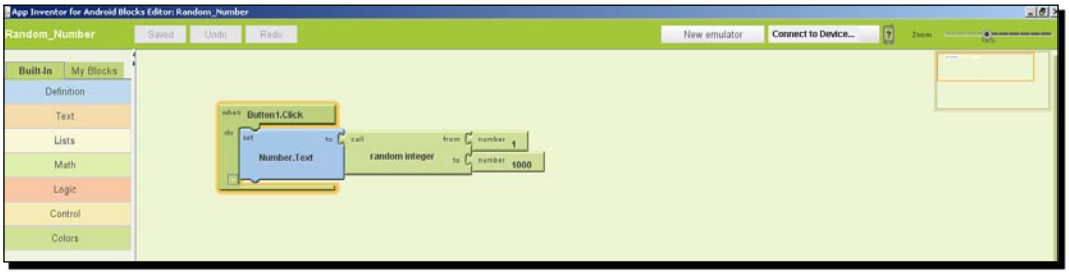

App Inventor
BETA
Search
Read the App Inventor Forums!
Forum > Table of Contents >
About
Learn
Forum
My Projects
App Inventor for Android User Discussion Forum
Read and follow these IMPORTANT GUIDELINES as you participate in these forums. This is the way to make many friends on the forums!
You can join one or all of these sub-forums. Click on the title of the sub-forum to see forum details and join. Read FORUM TIPS for more information on the functionality of these forums.
Read the USER FAQ, the TROUBLESHOOTING page, and the App Inventor DOCUMENTATION before posting. Chances are good your question will be answered.
Search the archives of these forums. Chances are good that somebody has run into the same issue before.
Do Not Report Bugs on these forums. Bugs and feature requests should be logged on the ISSUES LIST. The App Inventor team uses the data on the ISSUES LIST to inform its development priorities.
Be very clear about your question in the subject and give plenty of details in your post to help others understand your problem. Code snippets, screen shots, source files for your app may also be helpful.
App Inventor Announcements
Learn about the latest launches and feature updates for App Inventor. PLEASE NOTE: This sub-forum accepts posts from Googlers only. Users' posts are welcomed in the other App Inventor sub-forums. :)
Topics 3 of 6 View all »
A small update to the latest release
1 new of 1
App Inventor Team (1 author)
2/1/11
App Inventor has been updated.
1 new of 1
Mark Friedman (1 author)
1/28/11
Tell us about your educational use of App Inventor in 2010 and 2011
2 new of 2
App Inventor Team (2 authors)
1/12/11

My Projects: Which brings us back to the area where all the good stuff happens—our own projects.

Remember the little random number example program in the Preface? I showed you the blocks for it then. Well, below, in the Designer, is how I set the font size, color, and placement of the button (the components) of the app.



We'll be spending quality time during the course of this book in both the App Inventor **Designer** (previous screenshot) and the **Blocks Editor** (following screenshot).



Anyway, explore the Google App Inventor site at your leisure, but, for now, on with installing our working environment.

Requirements for Windows, Mac, and Linux

We need a computer.

Being based on the web and using Java (which works on most machines), App Inventor by design has all three of the major operating systems covered—Windows, Mac, and Linux.

Here are the specific operating system requirements:

- Macintosh (with Intel processor): Mac OS X 10.5, or 10.6
- Windows: Windows XP, Windows Vista, or Windows 7
- GNU/Linux: Ubuntu 8+, or Debian 5+

Web browsers are also important. App Inventor—while doing a lot of its work on our local computers—is controlled from online. Google recommends one of the following:

- Mozilla Firefox 3.6 or higher
- Apple Safari 5.0 or higher
- Google Chrome 4.0 or higher
- Microsoft Internet Explorer 7 or higher

I personally prefer and mostly use the new Mozilla Firefox 6. Google Chrome is nice too.

Obtaining and installing Java

Java is also needed since this enables the Blocks Editor to run on all the operating systems listed previously. You may already have Java on your computer; it's commonly found these days.

The presence or absence of Java is easily checked. Go to `http://www.java.com/en/download/testjava.jsp`. If the box below appears, you have Java and can go on to the next item of installing our working environment.

Installing Java on your computer

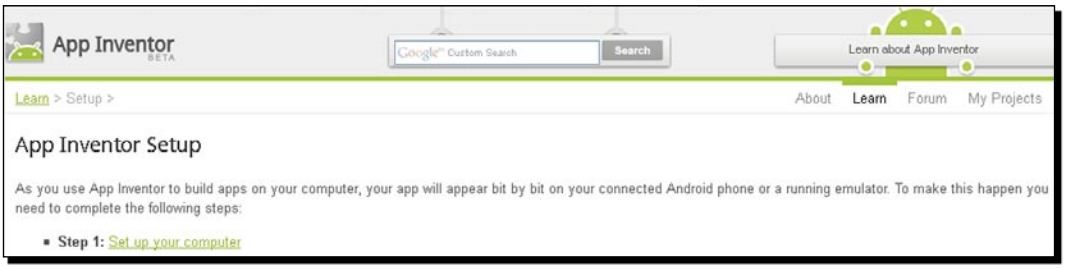
If the previous box on the Java site doesn't state **Your Java is working**, then follow the onscreen directions to install Java for your operating system.

Installing Java is the most time-consuming part of setting up App Inventor. The Java software is large (50 or 60 MBs) so requires a bit of download time depending on the speed of your Internet connection, but, as already stated, it's a one-time process.

After you can go to the Java test site and receive the positive message shown previously, continue on to the next section.

Installing App Inventor locally

For App Inventor to show our apps working in real time as we create them, it needs some software installed permanently on our local computers. To get this software, go to the following URL on the App Inventor site: `http://appinventor.googlelabs.com/learn/setup/index.html`


App Inventor
BETA
Search
Learn about App Inventor
Learn > Setup >
About
Learn
Forum
My Projects
App Inventor Setup
As you use App Inventor to build apps on your computer, your app will appear bit by bit on your connected Android phone or a running emulator. To make this happen you need to complete the following steps:
Step 1: Set up your computer

You must, of course, be logged in to your Google account before this web page displays. Near the bottom of this setup page, we find three links:

- Instructions for Mac OS X
- Instructions for GNU/Linux
- Instructions for Windows

Choose the one appropriate to your system. I'll shortly give you tips for all three.

Time for action – installing App Inventor on Mac

Click on the link for Mac OS X as shown previously on the setup web page. On the Mac installation page, follow the seven simple steps to install App Inventor on your Mac. The splash page that comes up after download is shown in the following screenshot, and the 1.1 installer is still current as of this writing, but, naturally, the version number is subject to change.

Install App Inventor Setup version 1.1
Welcome to the App Inventor Setup version 1.1 Installer
Introduction
License
Destination Select
Installation Type
Installation
Summary
This program installs files on your Mac that are needed for building applications with Google App Inventor for Android.
You will be guided through the steps necessary for installation.
This is version 1.1, released in October 2010
After installing, you can start creating Android applications by visiting the App Inventor web site at
http://appinventor.googlelabs.com
Go Back
Continue

One troubleshooting tip: in most cases, as is pointed out on the Mac installation page, App Inventor should be able to locate the installed software on its own. But, if it asks you to type in the location to look for it, the pathname to enter is:

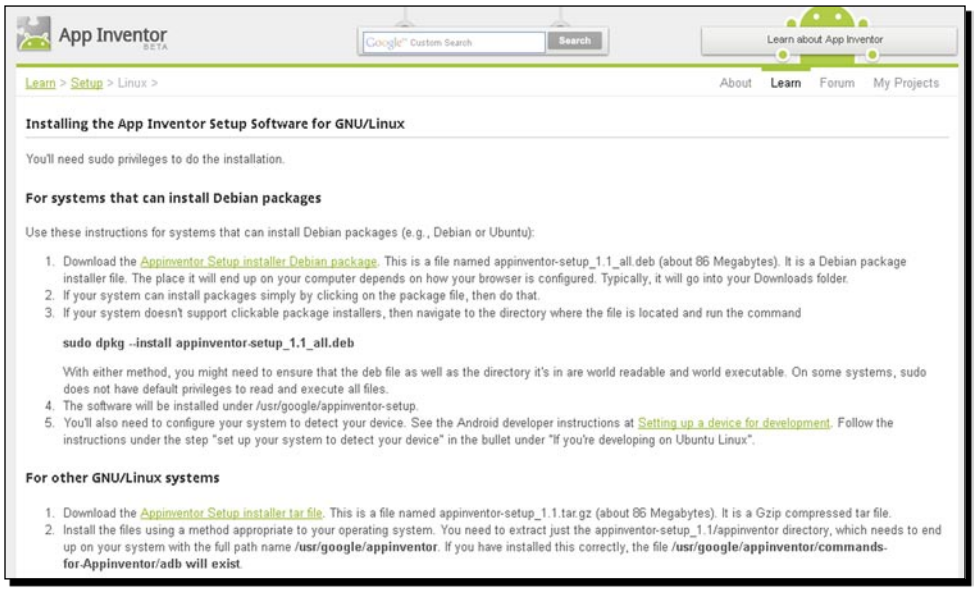
```
/Applications/Appinventor/commands-for-Appinventor
```

Mac folks may now skip ahead to the section about setting up and running the emulator.

Time for action – installing App Inventor on GNU/Linux

1. Click on the link for GNU/Linux as shown previously on the setup web page.
2. On the GNU/Linux installation page shown in the next screenshot, follow the instructions.

3. Please note there are two sets of instructions—one for Debian packages (such as Ubuntu) and a set for other types of Linux distributions.


App Inventor
BETA
Search
Learn about App Inventor
Learn > Setup > Linux >
About
Learn
Forum
My Projects
Installing the App Inventor Setup Software for GNU/Linux
You'll need sudo privileges to do the installation.
For systems that can install Debian packages
Use these instructions for systems that can install Debian packages (e.g., Debian or Ubuntu):
1. Download the Appinventor Setup installer Debian package. This is a file named appinventor-setup_1.1_all.deb (about 86 Megabytes). It is a Debian package installer file. The place it will end up on your computer depends on how your browser is configured. Typically, it will go into your Downloads folder.
2. If your system can install packages simply by clicking on the package file, then do that.
3. If your system doesn't support clickable package installers, then navigate to the directory where the file is located and run the command
sudo dpkg --install appinventor-setup_1.1_all.deb
With either method, you might need to ensure that the deb file as well as the directory it's in are world readable and world executable. On some systems, sudo does not have default privileges to read and execute all files.
4. The software will be installed under /usr/google/appinventor-setup.
5. You'll also need to configure your system to detect your device. See the Android developer instructions at Setting up a device for development. Follow the instructions under the step "set up your system to detect your device" in the bullet under "If you're developing on Ubuntu Linux".
For other GNU/Linux systems
1. Download the Appinventor Setup installer tar file. This is a file named appinventor-setup_1.1.tar.gz (about 86 Megabytes). It is a Gzip compressed tar file.
2. Install the files using a method appropriate to your operating system. You need to extract just the appinventor-setup_1.1/appinventor directory, which needs to end up on your system with the full path name /usr/google/appinventor. If you have installed this correctly, the file /usr/google/appinventor/commands-for-Appinventor/adb will exist.

As pointed out in the previous instructions, if App Inventor Debian installer package is downloaded, `sudo` privileges are required to run it. That means you must be a super user on the system, or otherwise request the systems administrator to install the software for you.

In the set of instructions for other Linux systems, the super user requirement is not mentioned, but since the software sets up under the `/usr` directory (which a regular user would not have write permissions in), administrator privileges are called for there as well.

If you are asked where the software is located, the directory path you should enter is: `/usr/google/appinventor/commands-for-Appinventor`

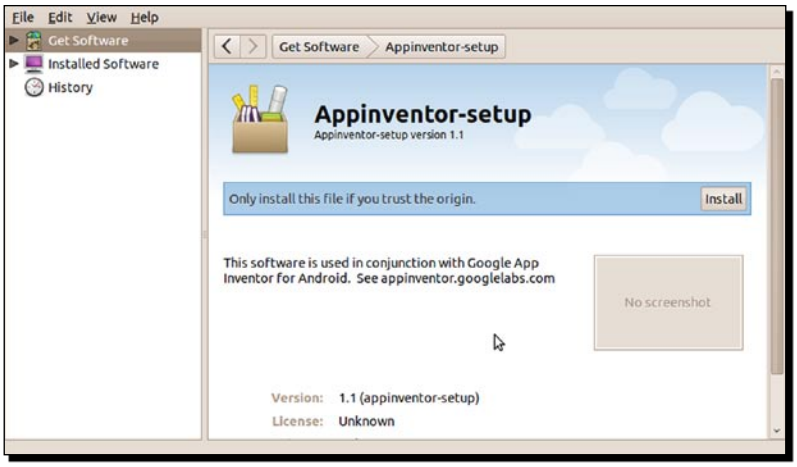
In my endeavors here (book publishing company, video production), I have several computers for my regular use spread over three buildings. So, I have Google App Inventor installed on four types of machines, one of which is an Ubuntu 10.04 Linux workstation.

I also use it on a Windows 7 machine, a Vista computer, and three XP-based computers. XP is still important in a production environment because some production software that is very important to my company does not yet have versions that live well in the newer Windows systems.

But, let's return to our Linux installation now in progress.

Download from Wow! eBook <www.wowebook.com>

Once the setup package is downloaded (this is on a Debian-based system, Ubuntu 10.04), we click on it and see the following installation screen. Just click again on **Install**, and the rest is automatic; that is, no setup configuration is required.

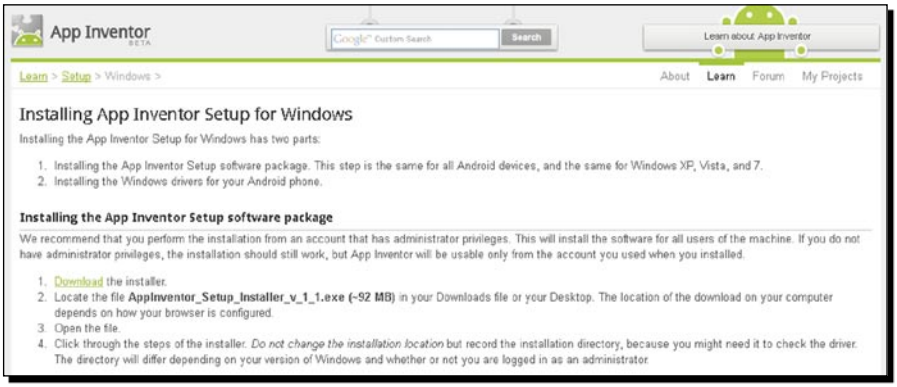


Once the setup completes (and it's quick), skip over the Windows setup info, and meet the rest of us at the section on installing the virtual phone emulator.

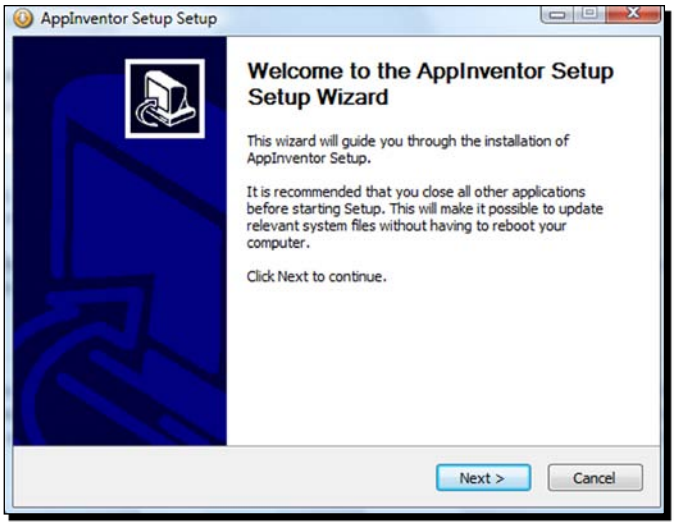
Should you have a non-Debian system, some manual setup might be required.

Time for action – installing App Inventor on Windows

1. Click on the link for Windows as shown previously on the setup web page for Google App Inventor.
2. On the Windows installation page, as shown in the next screenshot, follow the instructions as described on the instructions page shown earlier.


App Inventor
Search
Learn about App Inventor
Learn > Setup > Windows >
About Learn Forum My Projects
Installing App Inventor Setup for Windows
Installing the App Inventor Setup for Windows has two parts:
1. Installing the App Inventor Setup software package. This step is the same for all Android devices, and the same for Windows XP, Vista, and 7.
2. Installing the Windows drivers for your Android phone.
Installing the App Inventor Setup software package
We recommend that you perform the installation from an account that has administrator privileges. This will install the software for all users of the machine. If you do not have administrator privileges, the installation should still work, but App Inventor will be usable only from the account you used when you installed.
1. Download the installer.
2. Locate the file AppInventor_Setup_Installer_v_1_1.exe (~92 MB) in your Downloads file or your Desktop. The location of the download on your computer depends on how your browser is configured.
3. Open the file.
4. Click through the steps of the installer. Do not change the installation location but record the installation directory, because you might need it to check the driver. The directory will differ depending on your version of Windows and whether or not you are logged in as an administrator.

3. Once downloaded, click on the `Appinventor_Setup_Installer_v.1.1.exe` file (or newer version), and a splash screen appears. Follow the previous instructions on the web page to complete the installation.


AppInventor Setup Setup
Welcome to the AppInventor Setup Setup Wizard
This wizard will guide you through the installation of AppInventor Setup.
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Next >
Cancel

The Windows setup page mentions the various phone drivers that come with the App Inventor software. We'll discuss those shortly, but first, let's get to setting up and running an emulator.

What just happened?

Okay, now we've set up Google App Inventor to run on our various systems—Mac OS X, GNU/Linux, or Windows. But, how do we know if it's working?

Good question. We find out by using it! And the first thing we need to do is make sure we have the third component of App Inventor in place—an Android device to test our apps on.

Setting up and running the emulator

To do a quick review, App Inventor has three parts:

On the web: App Inventor **Designer**, the part that we access via our various Google accounts.

On our local computer: **App Inventor Blocks Editor** (downloaded from **Designer** by clicking on the **Open Blocks Editor** button), USB drivers for our devices (how to get and install those coming next), and emulator software (which we just installed as part of the App Inventor software).

A test device: That can either be the emulator on your computer or an Android phone, connected via an USB cable.

In my opinion, the ideal test device is a phone because that's mostly what we'll be developing apps for in this book. In my case, I use my Droid 2.

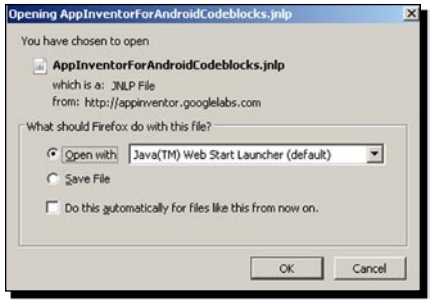
However, the emulator—software running on our local computer that gives us a virtual Android device to use for testing—is the first we'll look at. In fact, utilizing this virtual device, you can develop apps without even owning a phone.

Time for action – opening up the emulator

So, here's how you get to the emulator, which gives us a chance also to start learning our way around in App Inventor.

On the web, log in to your Google account (easiest way to find it is in the upper-right corner of `http://google.com`), and then go to `http://appinventor.googlelabs.com`, clicking on **My Projects** when you get there. You will already have projects, or you'll need to name one, and then will be presented with a new blank project-like as shown in the following screenshot:

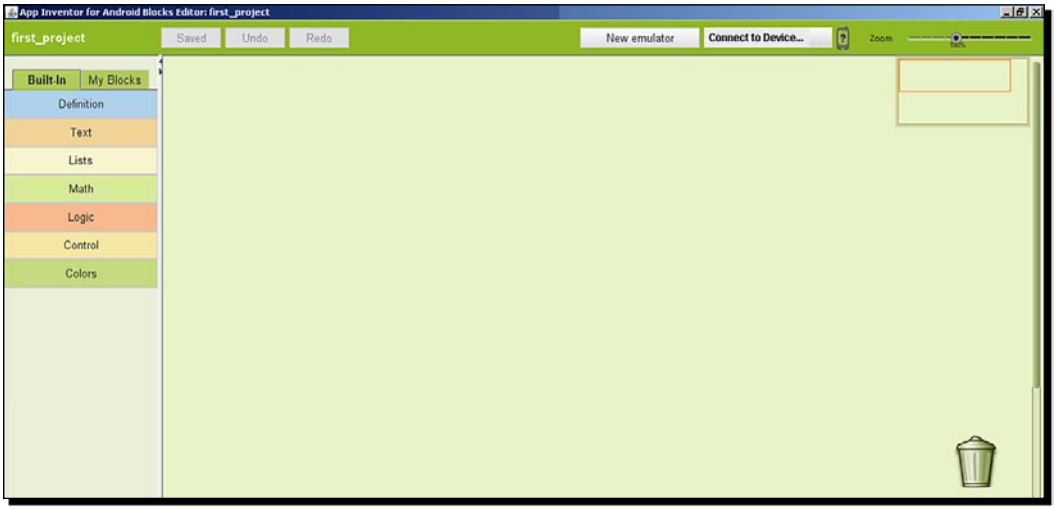
This is the way we will always enter App Inventor from the web, even though part of it is on our local computer. To open the portion of App Inventor that runs on your computer, click on **Open the Blocks Editor** on the top-right corner of the design page. We get a dialog box that looks something like the following screenshot (depending on your operating system):



Just click **OK** without making any changes, and a Java splash screen appears briefly, and then we get the bad boy we'll be spending most of our app development time using.

The App Inventor **Blocks Editor** (again, a Java application) looks like the one shown in the next screenshot. Center it up (a bit annoyingly to me, it always comes in partially offscreen) and we're ready to roll.

Some Firefox users get an issue of JNLP being saved rather than being opened. This can be fixed by going into Firefox **Preferences** | **Applications** | selecting **JNLP** and for action selecting **Use Java Web Start**.



If you now have the previous screen, congratulations!!—it means that all that installation of Java and the App Inventor software works, and you never have to worry about it again (okay, except maybe for the occasional update; we live in the world of software, after all).

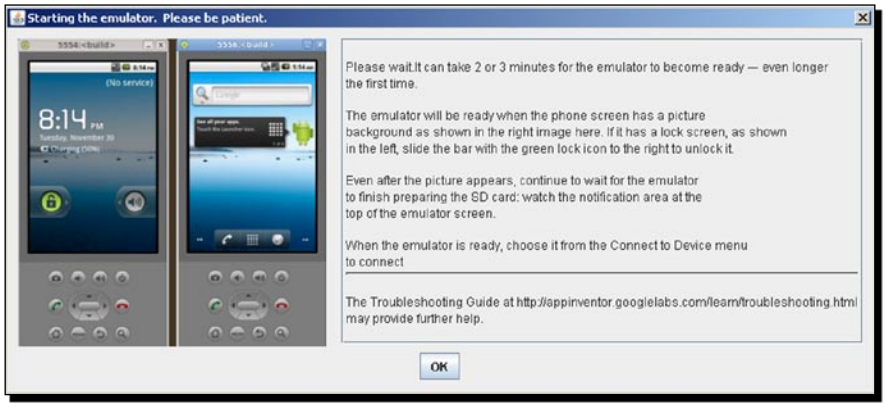
If the screen above does not appear, you need to troubleshoot your installation.

During the course of this book, we'll do tons and tons of stuff on this screen (the App Inventor **Blocks Editor**), but, for right now, all we want to do is bring up the emulator software and make sure it's there and working (which, almost certainly, it will be if you've gotten this far).

Click on the **New Emulator** button (upper-right of the previously shown Blocks Editor screenshot) and settle down with a good book (like this one, eh?) to wait.

Bringing up an emulator in App Inventor, or in other app creation systems such as the official Android SDK, will take just as long.

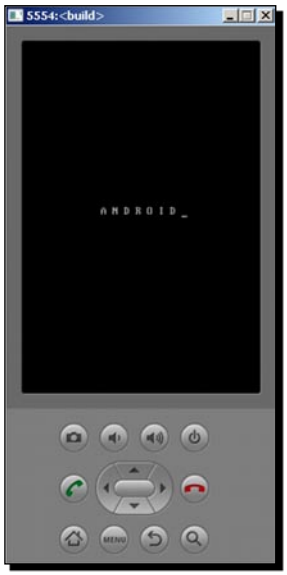
In fact, the first thing you see is a splash screen as shown in the following screenshot, which warns us precisely of this long delay:


Starting the emulator. Please be patient.
Please wait.It can take 2 or 3 minutes for the emulator to become ready — even longer the first time.
The emulator will be ready when the phone screen has a picture background as shown in the right image here. If it has a lock screen, as shown in the left, slide the bar with the green lock icon to the right to unlock it.
Even after the picture appears, continue to wait for the emulator to finish preparing the SD card: watch the notification area at the top of the emulator screen.
When the emulator is ready, choose it from the Connect to Device menu to connect
The Troubleshooting Guide at http://appinventor.googlelabs.com/learn/troubleshooting.html may provide further help.
OK

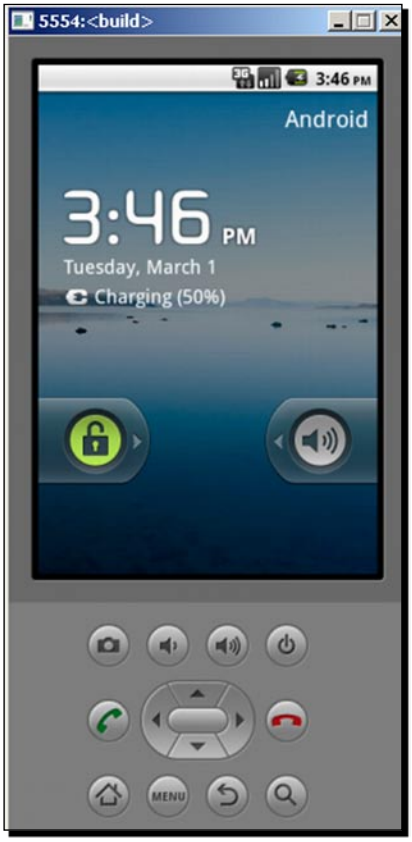
The operative wording here is even longer.

Because it takes so much time to bring up the emulator, keep it open for your entire work session, instead of stopping and starting it.

Don't get excited when the virtual phone screen, such as the one in the following illustration, seems to almost immediately pop up. There's still plenty of time to wait, this is just the beginning of the process.



Eventually, we see our virtual phone appear, and it looks like the following:

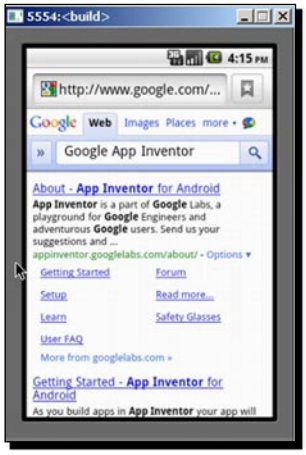


Unlock it just as we would with an actual phone by sliding the little padlock over all the way to the right. Use your mouse cursor, holding down the left button, instead of your finger in this case. Swiping on your computer screen only leaves fingerprints.

What just happened?

Once unlocked, we have most of the functions of a regular phone (as we see in the following illustration). This on-the-screen-phone won't actually make calls, but it can fake them for testing purposes, and so forth. Take a few moments and get familiar with this emulated phone.

But, this emulator is not all fakery. Type the search term `Google App Inventor` in the Google search entry at the top of the emulator's screen and click on the arrow that appears on the right. It goes out on the Internet and finds links just as you would expect. So, our emulator is pretty powerful and worth the wait.



The advantage of testing with the emulator is that we do not need a phone. The disadvantage is that it's very slow in both initialization and in operation!

Better to use your handy-dandy Android smartphone (although it, too, might be slow the first time you do this, but will be faster after that). But, how do we get it to talk to our computer? USB drivers, of course—let's find the ones we need.

Finding and downloading drivers

Drivers are bits of software installed on your local computer that enable a specific Android device to communicate with the App Inventor **Blocks Editor**. As stated earlier, our mobile devices—smartphones, tablets, and similar stuffs—work better and faster as test devices than the emulator.

However, for the device to be of use, it must be both configured correctly (that's our next section) and have a working link to the **Blocks Editor**. We can tell if our device is linked by clicking on the **Connect to Device** button at the top-right of the **Blocks Editor** screen. If one is present—connected to our computer via a USB cable and communicating properly—we'll see the serial number of the device, as shown in the following screenshot:

That happens to be my Droid 2.

The first stop, if you are having problems, would be the troubleshooting guide on the AI install page.

But, the problem here is that Android devices are proliferating rapidly. While some drivers come with the App Inventor software we installed on our machines, your device may or may not be supported. If it's not, you might have to do a bit of hunting to find a driver and get it working.

If you have a problem making your device communicate, chances are 14,000 other people have already had the same problem and solved it. Google (referring here to their search engine) is really your friend!

Driver problems for App Inventor are the same as for the Android SDK (Software Development Kit used widely in developing apps for Android devices), so doing a web search for driver issues on the Android SDK adds even greater chances of solving a specific driver issue.

Let's take a look at finding and configuring drivers on the three operating systems App Inventor works on.

Mac

Mac OS X systems seem to present the least trouble working with a wide range of Android phones. As one web poster said, **it just works**.

Linux

Getting our phones talking with App Inventor under Linux can be a bit touchy. Just as we have a lot of variants of phones, we also have many different Linux systems. Luckily, there are plenty of good folk out on the Internet who will help you.

I spent most of an afternoon getting my phone happy with App Inventor on Ubuntu 10.10.

Here's what finally worked for me in establishing a connection between App Inventor and my Droid 2.

You need `sudo` (super user) privileges and have your phone connected via USB cable.

Time for action – connecting a Droid to Ubuntu

1. On the command line, type `cd /etc/udev/rules.d`.
2. Create a one-line file: `SUBSYSTEMS=="usb",ATTRS{idVendor}=="22b8",MODE="0666"`.
3. Restart `udev` by issuing the command `sudo service udev restart`.
4. Type `cd /usr/google/appinventor/commands-for-Appinventor`.
5. Type `sudo ./adb kill-server`.
6. Type `sudo ./adb start-server`.
7. Type `sudo ./adb devices`.

Download from Wow! eBook <www.wowebook.com>

What just happened?

The last command shows us if the system now recognizes our Android device, as in the following screenshot:

```
ralph@taft:/usr/google/appinventor/commands-for-Appinventor$ ./adb devices
List of devices attached
0A3A4CBA1802A013        device
```

The alphanumeric sequence on the last line in the previous screenshot is the serial number of my Droid 2 (and any other phone or device will look the same). Seeing the system recognize a device like the one previous means you can now click the **Connect to Device…** button on the **Blocks Editor** screen, and use the device to test apps in real time as we build them.

We'll also be able to package apps (install completed apps) on the connected device.

But, what do those preceding steps mean?

Well, working backward, the `adb` commands (stands for Android Debug Bridge) are part of the App Inventor software we installed. It's a service and utility for connecting and testing Android devices to App Inventor. In steps 5, 6, and 7, we stopped the server and restarted it so that the new configuration file could be processed, then tested to be sure the device (my phone) was there.

Now, look back at step 1—that just took us to the `udev` rules directory (`udev` is an Ubuntu service that handles connections of dynamic drives such as CD or DVD drives and many other USB peripherals including smartphones).

Another good source for fixing driver problems is the *Getting Setup and Connecting Your Phone to App Inventor* section of the AI forums.

In step 2, we used a program editor (I use `emacs` because I like power) to create our one-line rule describing the USB connection to our phone (or, in this case, my phone). I named my file `10-motorola-droid.rules`.

Ah, but what if you don't have a Droid, or at least something besides a Motorola phone? You can, of course, use a filename reflecting your phone's manufacturer or model. And you'll have to change the USB Vendor ID. Motorola is `22b8`.

The following is a list of ID numbers for several manufacturers that might be helpful.

Manufacturer	USB Vendor ID
Acer	0502
Dell	413c
Foxconn	0489
Garmin-Asus	091E
HTC	0bb4
Huawei	12d1
Kyocera	0482
LG	1004
Motorola	22b8
Nvidia	0955
Pantech	10A9
Samsung	04e8
Sharp	04dd
Sony Ericsson	0fce
ZTE	19D2

By the way, you can have rules for more than one phone, and each device will be recognized whenever it is plugged in via USB cable.

So, while it might be a bit of a pain getting a Linux system to accept your phone, this only has to be done once unless you trade phones. Yeah, I want one of those new dual-core babies, too!

Windows

Some drivers for Windows are included with the App Inventor software. Specifically, there are drivers for:

- T-Mobile G1* / ADP1
- T-Mobile myTouch 3G* / Google Ion / ADP2
- Verizon Droid (not Droid X)
- Nexus One

All other phones, the App Inventor site tells us, will require obtaining and installing a driver elsewhere. The site adds: **Even with the phones listed above**, **it's been our experience that the Windows drivers do not always install automatically, and you'll need to do a manual driver installation**.

In case it's necessary to do a manual driver install, a good starting point for finding drivers is on the Google Android Developers website at `http://developer.android.com/sdk/oem-usb.html`. They list a table of links to sources from which you can download various manufacturers' drivers for manual installation (see the following screenshot):

OEM	Driver URL
Acer	http://www.acer.com/worldwide/support/ mobile.html
Dell	http:// support.dell.com/support/downloads/index.aspx?c=us&cs=19&l=en&s=dhs&~ck=anavml
Foxconn	http://drivers.cmcs.com.tw/
Fujitsu Toshiba	http://www.fmworld.net/product/phone/sp/android/develop/
Garmin-Asus	https://www.garminasus.com/en_US/support/ pcsync/
HTC	http://www.htc.com Click on the support tab to select your products/device. Different regions will have different links.
Huawei	http://www.huaweidevice.com/worldwide/downloadCenter.do?method=list&flay=software& directoryId=20&treeId=0
KT Tech	http://www.kttech.co.kr/cscenter/download05_ asp for EV-S100(Take)
Kyocera	http://www.kyocera-wireless.com/
LGE	http:// www.lg.com/us/mobile-phones/mobile-support/mobile-lg-mobile-phone-support.jsp
Motorola	http://developer.motorola.com/docstools/ USB_Drivers/
Pantech	http://www.isky.co.kr/cs/ software/software.sky?fromUrl=index
Samsung	http://www.samsung.com/us/support/downloads
Sharp	http://k-tai.sharp.co.jp/support/
SK Telesys	http://www.sk-w.com/service/wDownload/ wDownload.jsp
Sony Ericsson	http://developer. sonyericsson.com/wportal/devworld/search-downloads/android
ZTE	http://www.zte.com.cn/cn/ products/mobile/services_support/index.jsp

Time for action – configuring our device

Installing USB drivers, as we did previously, makes it possible for our computer and Android devices to see each other. Configuring our devices allows them to talk with App Inventor. And, here's the good news, this is the last step in setting up a Google App Inventor working environment.

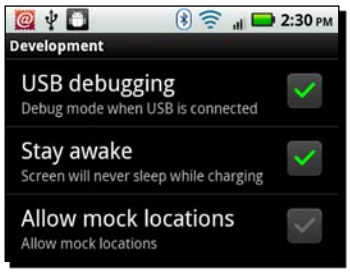
To reiterate, App Inventor's threefold method of creating apps uses Designer on the web, where we set up the appearance of our apps. Blocks Editor on our local computer lets us construct the logic (operation) of our apps by visually dragging and dropping blocks of code. And we test how well apps work by using an Android device attached to our computer, the latter being faster and easier than using emulator software.

Here's how we configure most Android smartphones, the screen captures being from my Droid 2.

1. Tap on the Home button (usually a little house under the bottom edge of the phone's screen).
2. Tap the **Menu** button.
3. Tap on **Settings**.
4. Choose **Applications**. We now should see something like the following screenshot:

5. Tap on the checkbox to the right of **Unknown sources** to get a green check mark as shown in the previous screenshot.
6. Move down the screen, and tap on **Development**.

On the **Development** screen, check **USB debugging** and **Stay awake** (these and the previous settings can be left in place from now on). Make sure the USB cable is not connected as it won't accept the check until it is unplugged.



7. Okay, tap your Home button, and return to your main screen on the phone. Then, connect the phone to your computer using the USB charging cable.

8. At the top of the phone's screen where the date, time, and other features are present (see the following screenshot), swipe on that bar with your finger to pull down the notification menu. You should see the **USB debugging connected** and **USB connection** items (and always update when you see that notification. I did).

9. Tap on USB connection, and the dialog box comes up as shown in the following screenshot. Tap on **Charge Only** and then **OK**. This procedure you'll do a lot because you'll want to return the connection to **USB Mass Storage** from time to time so that you can copy files to and from the phone.

What just happened?

Our testing device is now ready for use.

That completes setup of our devices for testing apps. We're now ready to create apps and learn all about App Inventor!

Pop Quiz

1. What are the three parts of App Inventor?
    a. Google.com, Microsoft Word, an iPhone
    b. Designer, Blocks Editor, an Android device
    c. Larry, Moe, and Curly
    d. None of the above
2. The blocks in Blocks Editor represent blocks of?
    a. Coffee code
    b. Visual C++
    c. Java
    d. Python code optimized for Android devices
3. Apps are?
    a. Small programs that show where you are
    b. Automatic replies to incoming texts
    c. Games and other fun things
    d. All of the above and a whole lot more

Summary

In this chapter, we signed up for the free Google App Inventor account and logged on to the App Inventor website.

We viewed the requirements for installing App Inventor on Windows, Mac, and Linux computers and (if we did not already have it) obtained and installed Java.

We then downloaded and installed the part of App Inventor that runs on our local computers and checked out the emulator software (a cute but fake smartphone on your local computer for testing apps).

Next, we found (if they were not already present) the device drivers (for our phone or other Android device and configured our device) to work with App Inventor.

So, we now have a Google App Inventor account and the underlying Java environment needed to run App Inventor's blocks screen on our computer and to create apps on our phone or other device, as well as package them for use.

Time to learn the basics so that we can design and create all those powerful apps we'll explore in the course of this book. The next three chapters cover those basics. First is the chapter on components.

2
Learning Components

In this chapter, we choose and use various components in designing our apps.

Components may be thought of as services. For example, basic components include buttons, labels, and checkboxes. Media components allow us to play sounds, show videos, and so forth. Social components let us interact with others by phone calls, e-mail, texting, Twitter, and so on.

Components are chosen and added to apps in the App Inventor Designer by dragging them from the Palette column and dropping them into the virtual phone screen in the Viewer column.

The virtual phone screen in Designer is not `WYSIWYG` (what you see is what you get), so starting Blocks Editor and having either an emulator or your phone connected is very useful in getting the design we really want.

We'll examine all of these component/services and see how they work, and why and when we would use them. And, most fun of all, we'll make some quick little apps as examples.

What we learn in this chapter:

- Using Google App Inventor's web interface
- Basic components
- Media components
- Animation components
- Social components
- Sensor components

- Screen Arrangement components
- LEGO® MINDSTORMS® Components
- Other components (brief mention, a lot more in *Chapter 4*, *Mastering Concepts and Advanced Components*
- Not ready for prime time components

So, let's get to it.

Using App Inventor's web interface

The part of Google App Inventor on the web, again it's called Designer, consists of only two pages: **My Projects** and **Design**.

As we learned in *Chapter 1*, *Obtaining and Installing Google App Inventor*, one gets to Designer by browsing to `http://appinventor.googlelabs.com` and, if not already on, logging in using our Google account. You will wind up either at the **My Projects** page as shown in the following screenshot or (if you already have a project in progress) in **Design**. Left-clicking on the name of a project opens it for editing.

One important concept here: App Inventor Designer may look like just a couple of web pages, but it's really a quite powerful web application. Keep in mind that you are running a program, not browsing a website, and that Google keeps your data (the designs and the blocks we'll click together later) backed up out there in the cloud (those zillions of servers they have all over the place).

But, "whoa, dude", you say. "Don't throw around buzzwords like cloud without some kind of explanation."

Okay, this one deserves definition because it is the whole concept behind Google App Inventor (and the many other web-based, therefore cloud-based apps) that Google and others are pushing these days.

Download from Wow! eBook <www.wowebook.com>

Cloud computing, according to Wikipedia, is Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on-demand, such as electricity.

What that means to us is that a lot more resources than we could ever hope to have on our local computer are packed into the cloud portion of App Inventor. We also get more timely updates and instantaneous access to new features as soon as they are implemented.

It's a good deal for us all.

My Projects

So, the first page (right, it's not a page, it's an application—sorry, good of you to catch that slip) is **My Projects**. Looking at the very top of the My Projects screen as shown next, we see first the App Inventor Beta logo. The fact that App Inventor is still in development is not, with Google, very significant. They tend to keep applications in development sometimes for years.

App Inventor
BETA
My Projects
Design
Learn
ralph.roberts@gmail.com | Report bug | Sign out
App Inventor was updated on Feb. 1
We updated App Inventor on Feb 1 to fix some problems with the release

Next, to the left, are three links: **My Projects** (the first of two parts of the **Designer** app), **Design** (the second part) we'll explore shortly, and **Learn** takes us to the page (and it is a web page, not an app) we met in *Chapter 1*, *Obtaining and Installing Google App Inventor*, which has all sorts of links to information and resources about and for App Inventor.

Finally, over in the far top-right corner is your (in this case my) account name, a link to report bugs, and a log-out link. Just below that but still above the big horizontal green bar is an info box where the nice Googlenauts (or whatever they be called, nice folk all) provide updates on the state of App Inventor such as updates, bug fixes, and so forth.

Now, back to the left end of that double-green decorative bar along the top portion of the **My Projects** screen. In the close-up shown in the following screenshot, we can see where most of the real action here takes place.

The **New** button lets us create new projects. **Delete** is a way to delete no-longer-wanted projects by clicking in the checkbox next to the project's name, then left-clicking on the **Delete** button. We use the **More Actions** drop-down menu to upload and download project files to and from **My Projects**. Let's look at these buttons' actions in detail.

To get to the **Design** part (we look at it in the next section) of App Inventor **Designer**, we must have an existing project or start a new one. First, here's how you create a project.

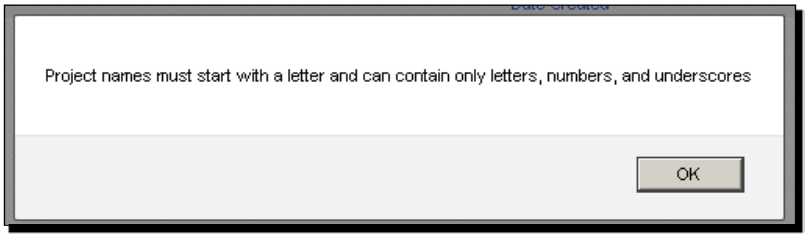
Time for action – creating a new project

It's really easy.

1. Click on the **New** button.
2. In the dialog box that pops up (like the one shown in the following screenshot), type the name of the project (no spaces or periods in the name, but you can use underscores) to be created and left-click the **OK** button.

Design opens up with our new blank project ready for design action. In the following section, we do things with our new project.

That is, a new project was created if we did not put any spaces in the project name. If we did, an error message pops up as shown in the following screenshot:



The Google cloud that we met earlier is probably mostly Linux-based servers as is common in big server farms (huge buildings full of huge computers humming away). Anyway, the Linux operating system does not like spaces in names. So `my spiffy new app` won't work as a project name but `my_spiffy_new_app` would. However, good practice is to keep the name as short as possible, as there's not much room to display app names on a phone.

What just happened?

We learned how to create a new project. We design these projects for a desired end result by the mix of components we choose. More about all that shortly (a lot more actually since this entire chapter is devoted to components, what they are, and where to put them).

Moving to the next button of those three on the green bar, we come to the middle one, the **Delete** button. Yes, it deletes projects.

Below the bar, if you have projects, they are listed. In my case (see the following illustration), I have a bunch of unneeded iterations of a current project, *Alien Names*. As I add new features to this app-in-progress, I like to save old versions for a while. This is very good programming practice, especially when we first learn a new system (like the one you and I both are doing with App Inventor).

For example, I thought things were going well (and they were) with the app getting quite sophisticated and able to generate literally trillions of alien names—App Inventor is really powerful! But, suddenly, whenever I hit the back button on my phone and exited the app, a big ugly error message came up the next time I tried to open my app. Definitely a not-ready-for-primetime moment.

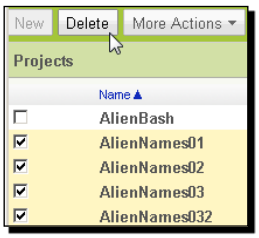
So, I checked back through my older versions until I found one that still worked correctly. I was then able to determine what had been added that generated the bug. *Bug*, by the way, is programmer slang for errors in a program. The term comes from back during World War II when the first huge vacuum tube computers were giving trouble, and it was found that real live insects (or usually fried insects by that time) had short-circuited some of the wiring.

In my case, it was a small lady bug that had somehow crawled into my Droid 2 and… Heh, heh. I'm kidding. No, it was misuse on my part of the **Canvas** component. Being a longtime publisher and graphic designer as well as a techie, I like my stuff to also look pretty. I was using **Canvas** to make lines and divide the app's screen, what it was meant for, and that created the error. So, instead, I now use **Label**— not what **Label** is meant for either but darned if it does not make some neat lines!

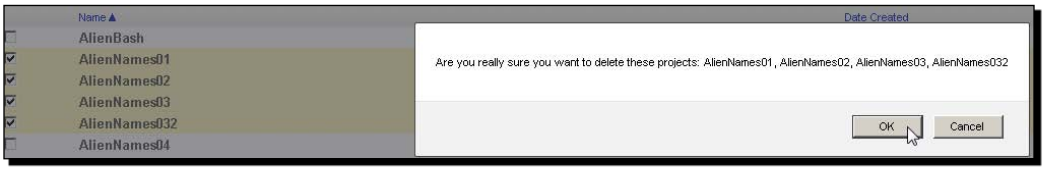
To make nice-looking dividing lines on your app's screen, use the **Label** component. Remove the text from it, set the background color to whatever color you want the line to be, set the width to Fill Parent, and the height to two pixels or any other thickness that looks good to you.

In later releases of App Designer, I've found **Canvas** works better and can be used for spacing and fancy dividers. We'll revisit this in a later chapter.

Anyway, the short of it is that I have old projects no longer necessary. So, here's how we use the **Delete** button, shown in the following screenshot. Check next to the names of projects for removal by left-clicking in the little box (the selection changes to a light yellow background). Left-click on the **Delete** button.



A "this is your very last chance" screen comes up like this one. Review the list of projects being deleted, and approve by clicking **OK**.



The files are now gone. But, what about the security of files you really want to keep?

The entirety of our App Inventor projects including design and blocks portions are stored for us (free) out there in the Google cloud. This is probably more secure than your or my local computer. Still, it does not hurt to have our own personal backups for those of us who like to be doubly sure (I raise my hand here).

More Actions gives us a way to download (get project source code out of **My Projects**) or upload (load source code into **My Projects**). You'll find quite a few sample source files (and lots of other great stuff) under the **User Generated Help Content** link on the **Learn page**. Left-click on the **More Actions** button as shown in the following screenshot, and you have choices to download or upload.

Time for action – downloading our projects

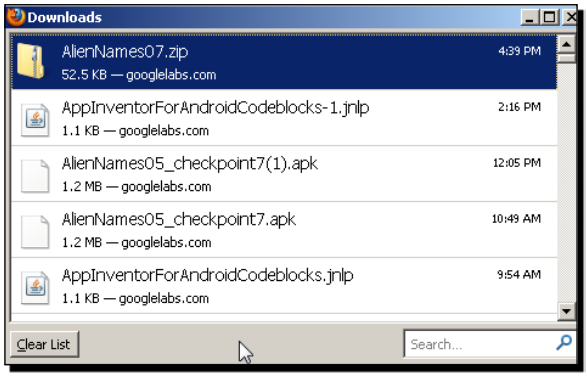
To download (save) a project to your local computer is simple but limited to one at a time.

1. Select the project to be downloaded by checking the box to the left of its name.

2. With the **More Actions** drop-down menu shown previously, left-click on **Download Source**.

The source code files are automatically compressed into a `zip` file and downloaded to whatever your web browser has set as its download directory on your computer. You can then move the files to wherever they are to be stored on the local machine.

Using Firefox in Windows, for example, press *Ctrl + J* on the keyboard to show the **Downloads** dialog box, shown in the next screenshot. Now, right-click on the file's name and choose **Open Containing Folder**. Copy or move the file from the resulting dialog box.



What just happened?

The project is now downloaded as a `.zip` file to our computer.

While we're here at **More Actions**, let's take a quick look at both—a good place to get sample source code and how to upload it into your **My Projects**. Check out this growing download list at `http://code.google.com/p/the-ai-repository/downloads/list` (the App Inventor Repository or *tAIR* as it is nicknamed, see also `http://www.theairepository.info/`). The source code, tutorials, block illustrations, and other information on that site is free for use (the top part of the download source code list is shown in the following screenshot):

In perusing that site in writing this section, I came across some code on e-mailing messages from App Inventor that looks interesting, so I've downloaded it to my computer. Now, let's bring it up into Designer.

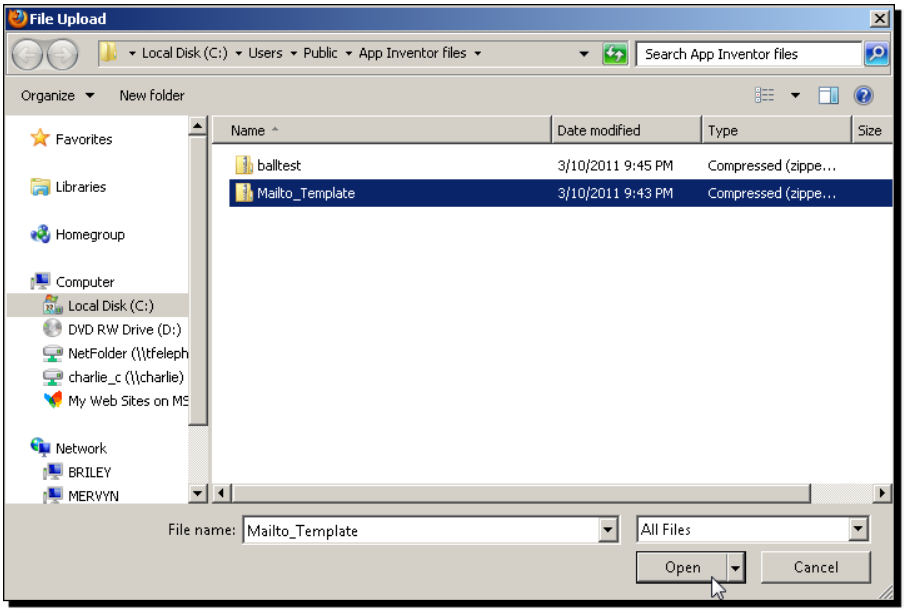
I suggest you make a directory just for App Inventor source code on your computer. That way, as you browse sites such as *tAIR*, you can grab interesting examples and quickly download them to a place where they will be easily found again.

Time for action – uploading source code

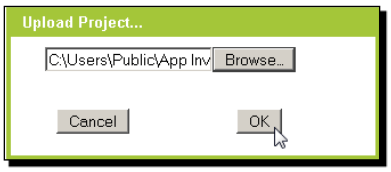
So, time to upload source code.

App Inventor source code is imported and exported as ZIP (compressed archive) files. You do not have to (and should not) unzip them; all that is handled automatically.

1. Click on **More Actions** and then **Upload Source**.
2. Choose **Browse** in the dialog box that pops up.
3. Find the source code ZIP file on your computer.



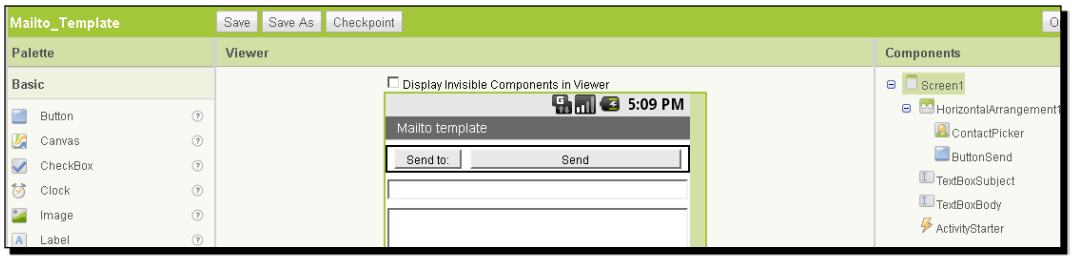
4. Select the compressed source file, and click on **OK** to upload it.



What just happened?

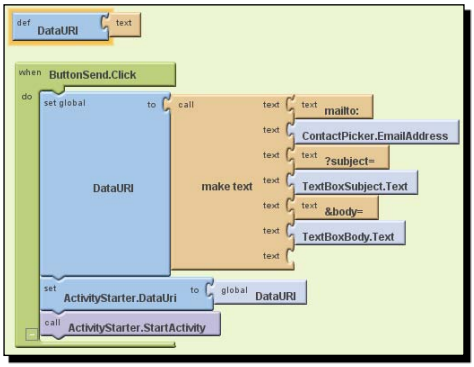
It was quick (source files are small). The file uploads, is saved as a project in **My Projects**, and immediately opens on the **Design** application page.

It is, as I had hoped, a simple example of how to send an e-mail from an App Inventor app; something all will probably be doing. The Designer portion of the source code, as shown in the following screenshot, is basically two buttons and two textboxes. The **Send to:** button brings up our e-mail contact list, and the **Send** button sends the e-mail. The textboxes are for entering the subject and body of the e-mail.



Ah, but what other goodies did we get in this upload? The blocks, of course, are what I really wanted to see. So, we open the Blocks Editor by clicking on the **Open Blocks Editor** button (as we did in *Chapter 1*, *Obtaining and Installing Google App Inventor*).

Here they are:



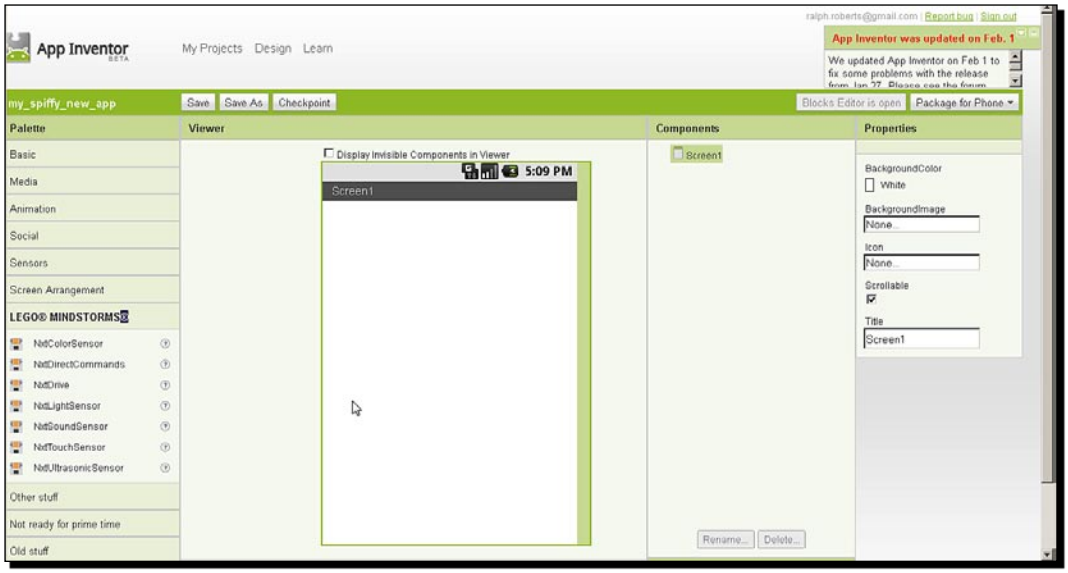
In total, we got a complete (if simplified) app that can be packaged and installed on our phone to let us pick someone from our contact list and send them e-mail. And, we can elaborate these blocks to fit our own needs. We'll look at stuff such as this in the next chapter, the one on blocks, but back to components.

That's about it for **My Projects**. Simple yet very powerful — it lets you create new projects, delete existing projects, and upload and download source code.

Now, on to the **Design** part of App Inventor Designer.

Design

Design, the second of the two screens making up the App Inventor Designer online application portion of Google App Inventor, looks like the following screenshot. We see this when we open a project in the **My Projects** screen that we explored just previously or log on when we have existing projects (App Inventor automatically opens in the last project you were working on).

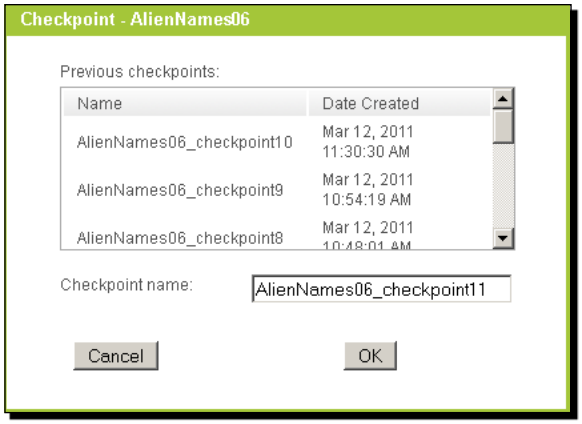


Looking at the very top white band (see the previous screenshot or—much better—log in and follow along on your computer), we note that it is exactly the same as the **My Projects** screen. It has the same links and the same information box over at the top-right edge. Consult the preceding **My Projects** section for descriptions.

However, the green bar is different. The following screenshot is a close-up of the left side of the bar. First, we note the title of our project is displayed. The next two buttons are rather obvious. **Save** records your project in its current state. **Save as** lets you save it under another name and that name, will be reflected on the leftmost position of the green bar because this new project is now open.

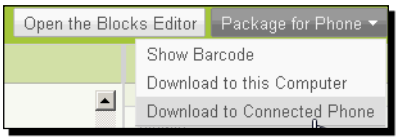
Checkpoint requires a bit more explanation. The **Checkpoint** button is similar to the **Save as** button, left-clicking on which saves a copy of the current project. However, it does not open the new project; we stay where we are but there is a growing list of these "checkpoint" versions of our project maintained as shown in the following screenshot.

Checkpoint files show up on the **My Projects** page as regular project names. Delete earlier versions when you are sure you'll not need them. But—as in my example of the *Alien Names* bug earlier—having these snapshots of previous iterations of your project can certainly be a lifesaver, rescuing you from having to spend time doing work over. Don't know about you guys, but I hate having to redo things.

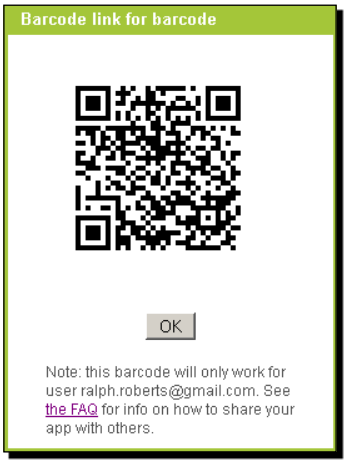


Now, slide all the way over to the right end of the dark green bar on the **Design** screen. It has two buttons, shown in the following screenshot. We've already met the **Open the Blocks Editor** button in *Chapter 1*, *Obtaining and Installing Google App Inventor*, while installing Java and so on. It starts up the **Blocks Editor**, the part of App Inventor that runs on our local computers.

The other button, **Package for Phone**, was also touched on in *Chapter 1*, *Obtaining and Installing Google App Inventor*. Left-clicking on it opens a drop-down menu allowing three choices as shown in the following screenshot. Let's take a quick look at these as we'll use them throughout this book, and you will use them for a long time to come in producing all your great Android apps.



The first option is **Show Barcode**. To use this first section and the remaining two, the **Blocks Editor** must be open (otherwise **Designer** can't package the app). Once we click on the **Show Barcode** button, the mini dialog box below comes up. This is a barcode link to download the app currently in **Designer** and install it on your phone. You will need a barcode reader app installed on your phone, but no cable connection is required.

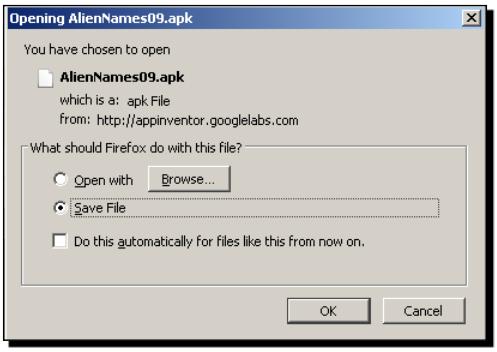

Barcode link for barcode
OK
Note: this barcode will only work for user ralph.roberts@gmail.com. See the FAQ for info on how to share your app with others.

With the barcode reader app on your phone, simply center the barcode on your computer screen in the app's reader window, and it will read a link like in the next screenshot. Tap on that link, and your app in Designer is downloaded to your phone and installed. You can then test it and make sure it works.

This link, as we are notified in the previous mini dialog window, works only if you are logged in to your Google account. But, you can open your **Design** screen and let anyone you like who has an Android phone or other device to install the app. But, this is very limited sharing, meant primarily for your ease in testing apps. Toward the end of this book, I'll show you how to package apps for the Android Marketplace and share them (even sell them) to the world!

Below is what a barcode scanner from a computer screen looks like on a phone using the Barcode Scanner app (which is free on Android Market). Note, it decodes the download link from the QR barcode. Tapping on this link installs the app.

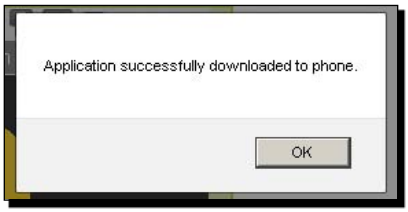
The second selection, **Download to this Computer**, zips the source code for your app into a compressed file, which includes all images, sound files, and other such files. It's then downloaded to your local machine, and you can share it with others who use App Inventor. This is the way the source code files we learned to upload and download just previously in this chapter were created. This takes longer, since App Inventor has to gather and package everything, but when a dialog box as shown in the following screenshot pops up, you can save this file on your local computer.



A file with an `.apk` file extension is an **Android Package** file, the standard format used for transmitting Android device apps. It's nothing esoteric—you can view the contents with any standard compression utility such as 7-Zip, Winzip, Winrar, or Ark.

Download from Wow! eBook <www.wowebook.com>

The third and final method of packaging an app is by downloading it directly to your phone or other device. In this case, it must be physically connected to your local computer via a USB cable. Once you click **Download to Connected Phone**, small notices with yellow backgrounds appear at the top center of the **Design** screen. These inform us that App Designer is first packaging then downloading our app to the connected phone. It takes about a minute or so, then we get a small dialog box letting us know the app is on the phone and ready to run.

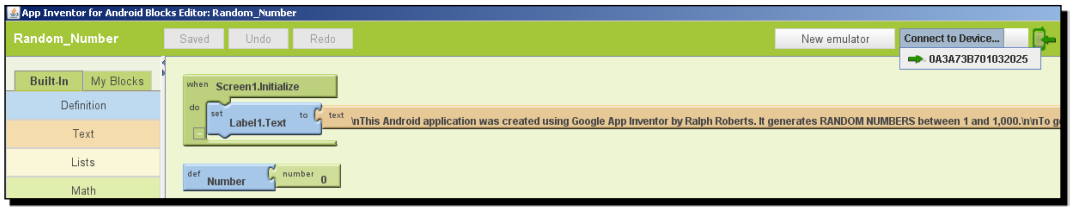


Again, for any of these three methods of packaging an app to work, **Blocks Editor** must be open. Even though the **Design** part of App Inventor Designer is where we package from, the connection to your phone is achieved in **Blocks Editor**. If you do not have a connection, **Design** lets you know by a red background error message that appears top-center on the **Design** screen (same place the yellow background packaging-in-progress messages show). Here's what the no connection error message looks like:

To quickly review what we learned in *Chapter 1*, *Obtaining and Installing Google App Inventor* about making a connection with an Android device, do this. Let's see how we do that.

Time for action – connecting our phone

1. On the **Design** screen, check to make sure the Blocks Editor is open (Design helps you out here, since the button says "Blocks Editor is open" if it is). If not, then open it.
2. Now, switch over to **Blocks Editor**. You probably have a small Java icon at the bottom of your computer screen. Just click on it if so—if not, rotate around through your open programs (*Alt* + *Tab* on Windows systems, for example) until you find it.
3. In the **Blocks Editor**, make sure your phone is plugged into the computer via a USB cable, and left-click on **Connect to Device**. A drop-down list shows the available devices (that is, the identifying serial to my Droid 2 in the next screenshot). Just click on the device serial and it connects.



You'll know the phone is connected with App Inventor when the app you're working on appears on the phone.

If you don't get a connection, the most common problem is not having the USB connection on your phone configured for **Charge Only** (see my phone's screen):

To get this setting screen on Droids (as an example), pull down on the bar across the top of the phone's screen, press on **USB connection**, and tap on the circle next to **Charge Only** to select it. Change it back to USB Mass Storage whenever you need to transfer files between your phone and computer.

Your Android device may look somewhat different—I recently read there are over 70 brands now and growing—but the principle is the same.

What just happened?

Our phones are now connected to App Inventor, which allows us both to install our apps and to use the phone as a test device in real time.

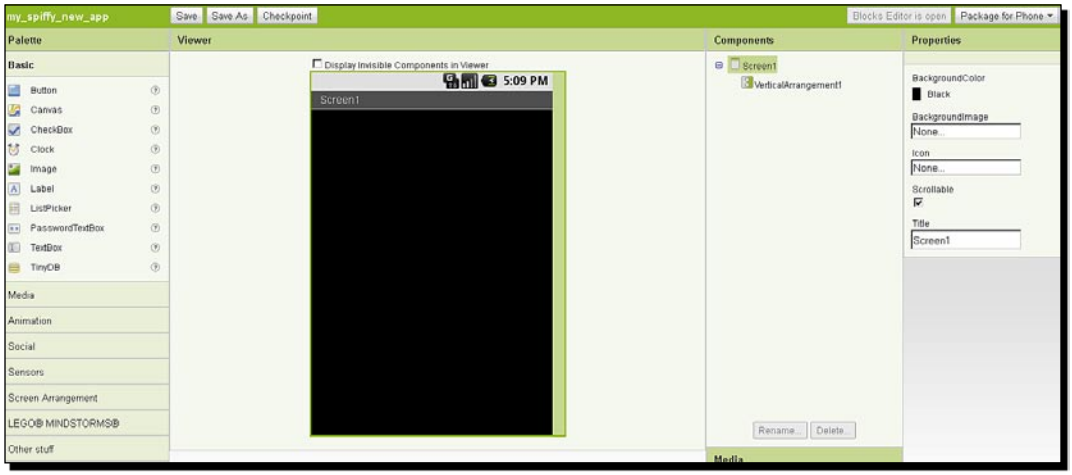
"In real time" means when we make a change in either **Designer** or **Blocks Editor** (for good or bad), we see it on our phone.

Designing our apps

Okay, so far we've talked about creating projects, deleting projects, uploading and downloading source code, saving projects, and saving checkpoints as we configure and build our projects.

The remainder (after this section) of this chapter is an overview of all the components (services) currently offered in App Inventor. As we explore components, we'll also learn more about what the rest of the **Design** screen/application does.

This (the lower or majority of **Design**, see the following screenshot) consists of four columns: **Palette**, **Viewer**, **Components**, and **Properties**.



We first met this very important area in *Chapter 1, Obtaining and Installing Google App Inventor* (and used it in designing our `NoTextWhileDriving` app). Here's where all the design action occurs.

It works like so: **Palette** is our library of components. We drag a component we want to use over to **Viewer** and drop it on the virtual phone screen.

In **Components** we can rename components to avoid confusion, or delete unneeded ones.

It makes life much easier if you give components a unique name based on what they are as soon as they are dropped into place. Instead of having 40 buttons named "Button1, Button2, ... Button40", it's simpler to name them after the function they perform when pressed, such as `btn_send_btn_email` or `btn_play_btn_sound`. And putting an identifier as to the type of component groups them in one place in the **My Blocks** column. You should apply this rule to all components.

Finally, on the far right, we have **Properties**. This category allows us to specify foreground and background colors, sizes, text, and various other items depending on the purpose of the component.

We'll learn more about these four columns as we examine individual components during the remainder of this chapter.

Another brief review to set the stage; Google App Inventor consists of three parts:

1. App Inventor Designer, which has two screens—**My Projects** and **Design**. This chapter is covering those two online application screens and the **Palette** (library) of components provided.
2. App Inventor Blocks Editor, a Java application that loads onto your local computer but is controlled by the online portion of App Inventor. There, we drag about blocks of code, instructing our components what functions to do and how to react—programming visually without having to even see all those cryptic lines of code.
3. A test device, such as your phone or some other Android device or a software emulator on your local computer.

App Inventor remains very much in a state of development, so expect new components to be added from time to time. These and other upgrades to App Inventor appear automagically (without any action by us) in App Inventor. This latter is one of the great advantages to using an online application.

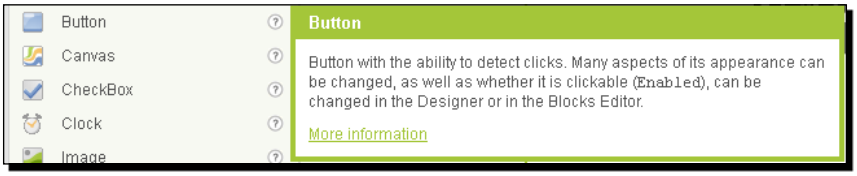
Basic components

The leftmost column, **Palette**, stores components. If you need one button or the 40 alluded to above, drag-and-drop them into the virtual device screen under **Viewer**. **Design** automatically numbers them in sequence. We, of course, will immediately rename them in the **Components** column (I'll show you how in a moment).

The **Palette** column has "drawers" of components, grouped by categories. This section is about the Basic group, which has the simplest but most used components.

If you have not already, click on **Basic** to open up that category, which now looks like the following screenshot:

Next to each component choice (to the right) is a small circle with a question mark (**?**) in it. Click on this for a help message about that component, which in the case of our first component, **Button**, looks like the following screenshot:

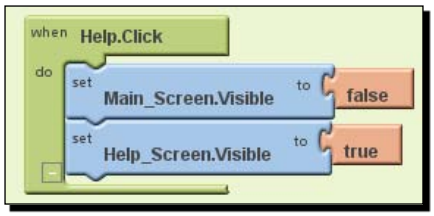

Button
Canvas
CheckBox
Clock
Image
Button
Button with the ability to detect clicks. Many aspects of its appearance can be changed, as well as whether it is clickable (Enabled), can be changed in the Designer or in the Blocks Editor.
More information

Click on **More information** in the mini dialog box above for additional info. But, for now, let me show you some cool stuff about the **Button** component.

Button

The **Button** component at first glance appears simple. In an app, the user taps on it and something happens. Simple? No. A **Button**, like most App Inventor components, provides a framework where a lot of stuff can occur. We'll see this in greater detail in the next chapter, the one on blocks.

To help us visualize it, here's an example of a button as it might be used in the Blocks Editor where we give our chosen components their powers:



This is a very rudimentary example from my published app, *Alien Names* (available on Android Market). It's the **Help** button. Tap on it, and the app turns off the main screen and turns on the help screen. That's nothing, I have buttons in that app that do 30 actions or more in randomly generating names.

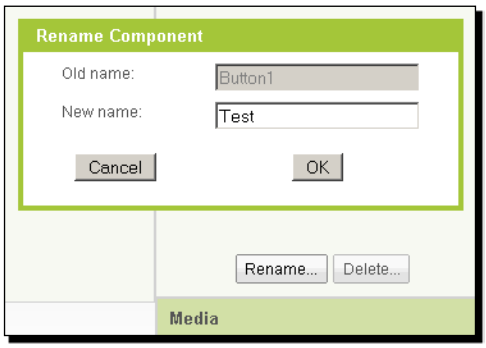
But, you see how we will be using the components, once they are in place and we've done a bit of design work in laying out our screen, choosing colors, and putting text in tables. Let's start by putting a button in place.

Open up a new project or the `my_spiffy_new_app`, the one we created earlier in this chapter.

Time for action – adding and configuring a button

1. In the **Palette** column, left-click on **Basic** (if it is not already showing the list of **Basic** components). Then, place the mouse cursor over **Button**, and hold down your left mouse button.
2. Drag the **Button** component over the virtual phone screen in the **Viewer** column, and release the mouse button. The result should look like this (and, by the way, it's always 5:09 PM on that "phone"):

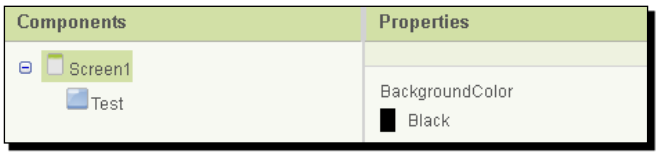
The first thing we should now do is rename the button to reflect its purpose. Let's rename this one to **Test** and use it in a little `Test` app. At the bottom of the Components column, click on **Rename**. In the resulting mini dialog box, type `Test` in the **New name:** entry box. Left-click on **OK** as shown in the following screenshot:



That changes the name of the button—making it easy for us to follow what's what when we have lots of blocks.

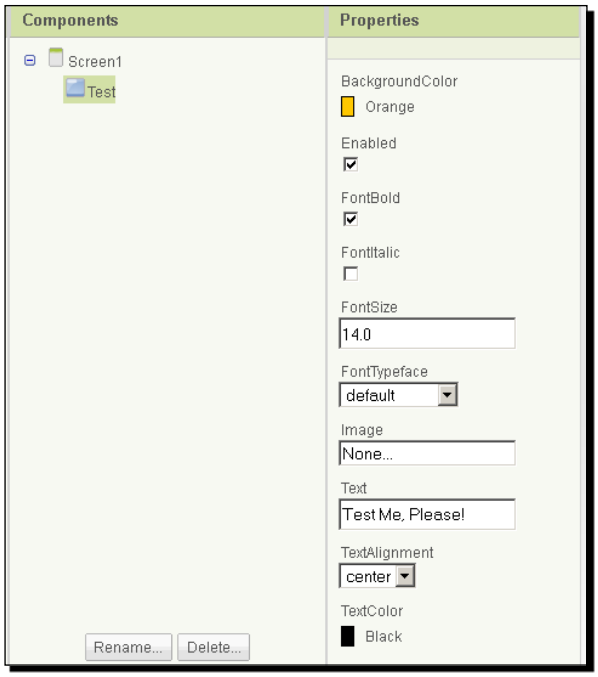
Now, time to use the **Properties** column to "prettify" our little app in progress. For this, we most often use the **Properties** column, but first select the component for modification by clicking on it in the **Components** column.

In our case, we only have two components so far—**Screen1** (which is there by default) and our newly renamed **Test** button (both depicted in the next screenshot). By the way, there is currently only one physical screen possible in App Inventor, but, not to worry, I'll show you later how to make as many virtual screens as you like! The end effect will be the same. We will start with **Screen1**.

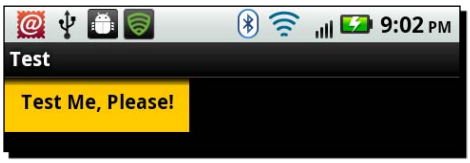


In the **Properties** column, change the **BackgroundColor** to **Black** by clicking on the color box and choosing that color. No particular reason for this. I just think it looks snazzier than the default bland **White**.

Now, let's configure our button. Left-click on **Test** in the **Components** column and then look at the **Properties** column, as shown in the next screenshot. The **Button** component, whatever we name it, has more configurable parameters than the screen component.



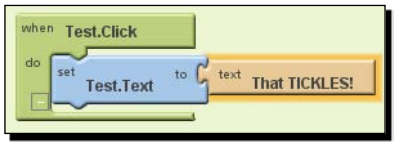
Why not make the background of the **Test** button **Orange** (it just looks good on black, click on **BackgroundColor** to do the change). Click **Font Bold** for an easier-to-read label on our button. Finally, in the **Text** box, put `Test Me, Please!`, and we have now finished the design of our app.



Just to follow through the entire process, go ahead and open Blocks Editor and connect to your phone or the virtual emulator. This is what the screen looks like. Take your time to not only craft apps that run well but also make them cosmetically pleasing. Nothing screams amateur louder than to leave the default values in place, eh?

Now, we'll complete the button's operation in the Blocks Editor.

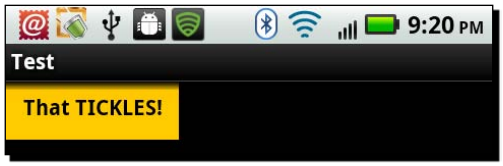
Under **My Blocks**, go to **Test** and drag out the **when Test.Click do** block (which describes what the button does when it's pressed). That's our framework for the button's operation logic as shown in the next screenshot. Drag out (from the same **Test** drawer) **Test.Text**. Click it into place. And, under **Built-In/Text**, get a **text** variable (the top choice), drag it out, and click into place as follows. Change the value of the text variable by clicking on the default text and editing it to **That TICKLES!**, and our program is done.



What just happened?

We just completed our second app (the first was `NoTextWhileDriving` in *Chapter 1*, *Obtaining and Installing Google App Inventor*).

Of course, since we have our phone or an emulator running, everything is live! Click on the button on your device. As shown in the following screenshot, the text on the button changes to **That TICKLES!**

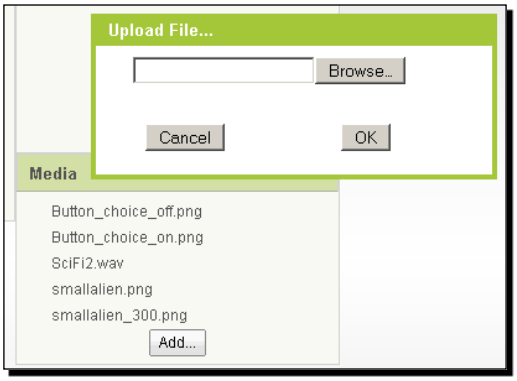


We changed the name of our **Button** component on the **Design** screen to **Test**. If we change it again there, all the associated blocks and the drawer name under **My Blocks** change to reflect that, as well as inside all the groups of blocks it's used in without breaking your app! Is App Inventor cool or what?

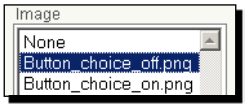
Go back to the **Design** screen now, please.

And, speaking again about "cosmetically pleasing", one way to really spiff up our apps is by using images for buttons! An example is in my **Alien Names** app (detail of main screen shown next). I created these in Adobe Photoshop, but there are tons and tons of free button images out there on the Internet for website designers, and they work quite well as fancy buttons in App Inventor also.

To load images and other media files, look for the **Media** section in the **Components** column of **Design**. It lists the files already uploaded and available for use. Left-click on the **Add** button and the **Upload File...** mini dialog box appears. Choose your file and click **OK** (see `http://developer.android.com/guide/appendix/media-formats.html` for all the formats AI accepts).



Now, making sure the button you wish to change to an image is selected in the **Components** column, shift over to the **Properties** column and click in the **Image** box. A drop-down menu of all media files shows. Choose the image you want as shown in the next screenshot.
You can have images, as I do, for when the button is "off" and "on." In Alien Names, the orange-colored buttons are not selected and, when selected, the button turns green by using the second image. Using images can make your app quite large in size, so you should watch for that when using a lot of images.



Sizing the buttons and the overlay text is done the same way whether you use the default button or an image.

Buttons are certainly the most used component in App Inventor. To sum up, it performs an action when the user taps it. We can control (design) its appearance (size, color, use an image) in **Design** (we can also reprogram them to appear differently later in our app). In Blocks Editor, we program what those actions are and how they work.

As we go through this book—which, after all, is meant to be learning Google App Inventor by example, we'll be doing a lot more in changing how buttons look and how they operate.

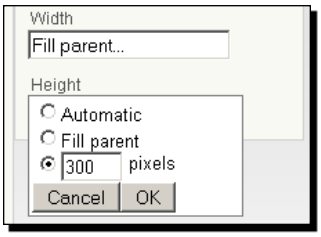
For now, let's buzz through the rest of this chapter and introduce the other components to you. Then, we get to play with blocks in the next chapter! Believe me, it is tremendous fun.

Canvas

The Canvas (like a painter's canvas on an easel) is a two-dimensional rectangular area on which drawing can be done and sprites (animated elements) move. The canvas is the basis of many games. To understand it and have a little fun, do this:

Time for action – follow the bouncing ball

Start a new project, name it ... say ... `Canvas`. Drag a **Canvas** component onto the virtual phone screen. In **Properties**, change its **Width** to **Fill parent...** (makes it as wide as the phone's screen) and the **Height** to `300` **pixels** as shown in the following screenshot:

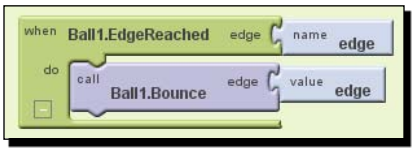


Now, just to jump ahead for a moment; under **Palette**, click on the **Animation** drawer and pull out a **Ball** component. Drop it on the Canvas on the virtual phone screen.

In the **Properties** column for **Ball1**, set **Heading** to `30` (bounces at an angle), **Interval** to `100` (moves more often), **Radius** to `30` (larger, easier to see), and **Speed** to `30` (moves faster). That completes the design portion.

Go to **Blocks Editor** (getting used to bouncing around like our ball, eh?).

From **My Blocks**/**Ball1**, pluck out **Ball1.EdgeReached** as shown in the following screenshot. From the same drawer, get **Ball1.Bounce** and click it into place. Then, from **My Blocks**/**My Definition**, grab **Value edge** (automatically generated for you) and click it into **Ball1.Bounce**. App done! Look on your connected device, and the black ball bounces incessantly on the white **Canvas**. Hit the back button on your phone to exit.



Animation is one of the things **Canvas** enables. We'll see others by example throughout this book.

What just happened?

A couple of points in general about the blocks here; the little holes or slots blocks plug into are called **sockets**. AI helps us out by putting up an error message if we try to plug in a block that will not work with the current block.

Also, when the blocks are automatically created by our choice of components in **Design**. When we go to **Blocks Edit** and look in the drawer of blocks for that component (in **My Blocks**), we find blocks are often paired; that is, there's one that is used to get value and the other that is used to set value. We choose the one fitting the task at hand.

Now, back to the basics.

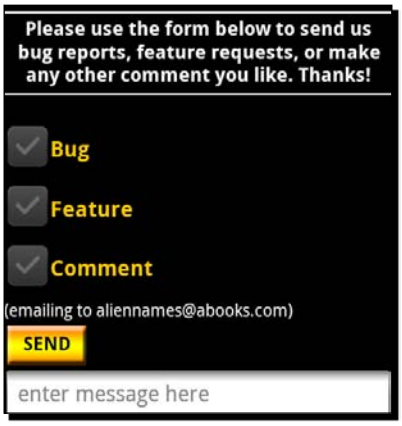
Checkbox

Checkboxes you already know about—if you have a phone or other Android device, tap them to put a check in the box or tap again to remove the check. They are widely used to set or unset options.

Checkbox the App Inventor component lets us add and program checkboxes to our apps. Checking or unchecking one of these returns a Boolean result—`true` or `false`.

Set the properties of **Checkbox** and all other components in the **Properties** column of the **Design** screen.

Here is a use of checkboxes from my `Alien Names` app (the checkboxes let the user choose the type of message they want to send):


Please use the form below to send us bug reports, feature requests, or make any other comment you like. Thanks!
Bug
Feature
Comment
(emailing to aliennames@abooks.com)
SEND
enter message here

Clock

The **Clock** component deals with time. We can use it as a timer, causing events to occur at regular intervals. It also converts and allows manipulation of time units. It is a powerful component, and we'll take time (pun, as ever, intended) to look at all its functions in the next chapter.

Clock is an invisible component. When you drop it onto the virtual screen in the **Viewer** column, it drops below the screen and does not show in the apps we make, but rather provides actions we can manipulate in the App Inventor **Blocks Editor**.

Image

An image is a photo or other graphic element to be inserted into our app. Once uploaded (use the **Media** section in the **Components** column of **Design**), the image may be used as often as needed. See again the Android programming site referenced earlier for more on formats, but AI essentially allows us to use JPEG, BMP, GIF, and PNG image files.

Label

While the **Button** component is the most widely used in App Inventor, certainly **Label** runs it a close second.

The **Label** shows text, which may be specified in the **Text** in the **Properties** column of **Design** or (as we shall do often in this book) be changed on the fly by our app (we set that up in the **Blocks Editor**). Other properties, all of which we can modify in either **Design** (which again is the second screen of the App Inventor **Designer** online application) or **Blocks Editor** to implement logic and control the appearance and placement of the text.

Download from Wow! eBook <www.wowebook.com>

Perhaps this will come as a surprise, but another use of **Label** is for lines. Google App Inventor—still being in beta testing stage—is short on some components that improve the look of your app, one of these being a way to draw lines on the screen.

No problem. Label also works quite well to make lines with. In the following screenshot, my `Alien Names` app, you see a **Button** and three **Label** components in use.

The **Button** and the title of the screen are in a **Horizontal** screen arrangement (we'll get to that, it lets you group components side by side). The three **Label** components below that are in a **Vertical** screen arrangement (allows components to be placed one over the other).

To get the two **Label** components I use for lines to become lines, I remove the placeholder text in **Text**, set their **BackgroundColor** to **White**, **Width** to **Fill Parent**, and **Height** to **2** pixels. Looks good, I think.

And, for a vertical line, it would be **Width** to **2** pixels (or whatever thickness you like) and **Height** to **Fill Parent**. We'll do some examples of this as we design our apps.

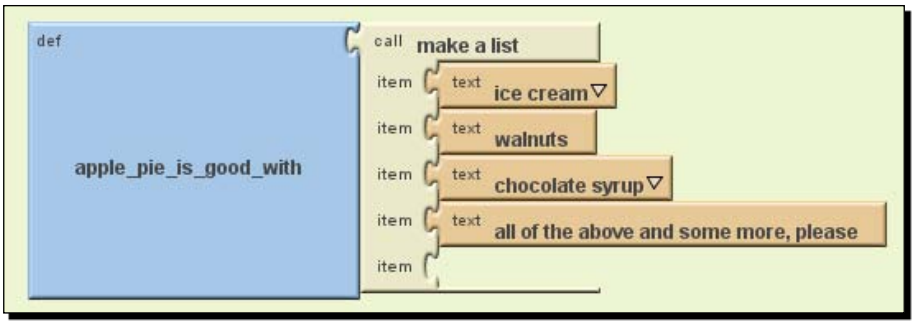
1:50 PM
Alien Names
Home
Bug Reports / Comments
Please use the form below to send us bug reports, feature requests, or make any other comment you like. Thanks!

To become more familiar with the design process, drop these elements in the virtual screen in **Design** and play with changing their properties in the **Properties** column.

ListPicker

In **Design**, we drag-and-drop the **ListPicker** component onto the virtual phone screen so that we'll have the tools we need in **Blocks Editor** to manipulate lists by retrieving elements and doing something with them. It's quite powerful, and examples of pulling and using items from lists come in later chapters.

Lists of items are set up in **Blocks Editor** by defining a variable with the name of the list and pulling out the make a list block from **Built-In**/**Lists**. Below, we fill the list with text variables (get them for free from **Built-In**/**Text**).



PasswordTextBox

The **PasswordTextBox** component lets the user enter a password with the text hidden (loose lips may sink ships, but prying eyes can break into your bank account). You've seen these a zillion times (see the next screenshot). The dots replace the actual letters of the password as they are typed.

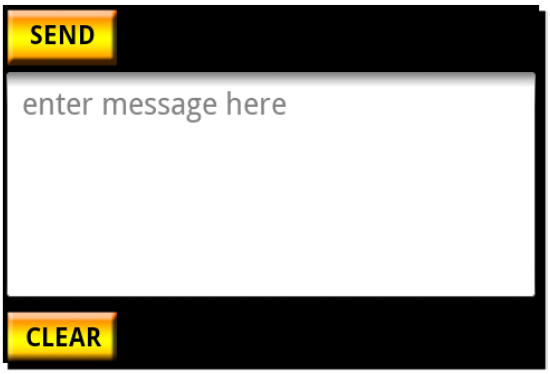
We use blocks in **Blocks Editor** to set up processing the password such as letting the user proceed if it is correct.

A password textbox is the same as the ordinary **TextBox** component (which we'll meet in just a moment), except that it does not display the characters typed by the user. So, there are other uses for it as well. Like anything of a sensitive nature where only the person tapping (you tap with your finger on a phone, you type on a keyboard) in the text needs to know what it is.

TextBox

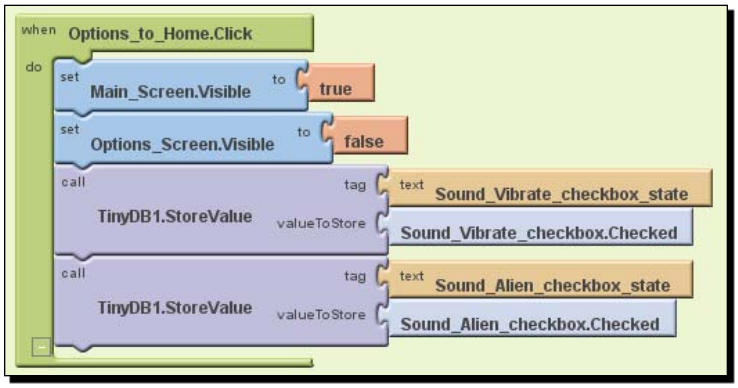
The only real difference between the **TextBox** component and the **PasswordTextBox** we just looked at is that you can see the text as it is typed into a **Textbox**. Textboxes allow the user to enter text in small to large quantities.

An example would be for the subject and body of an e-mail; you would typically use a button to (as in the example below) send the message and then clear the box so that the next message might be entered.



TinyDB

The **TinyDB** component (tiny database, another invisible item) stores values we want to be persistent (be remembered by the app and available the next time it runs). Here's an example (following), again from `Alien Names`. I included vibrate and/or an alien sound option for when buttons are pressed. The **Options** screen in the app provides a way to turn these on or off.


when Options_to_Home.Click
do
set Main_Screen.Visible to true
set Options_Screen.Visible to false
call TinyDB1.StoreValue tag text Sound_Vibrate_checkbox_state valueToStore Sound_Vibrate_checkbox.Checked
call TinyDB1.StoreValue tag text Sound_Alien_checkbox_state valueToStore Sound_Alien_checkbox.Checked

Here's how it works:

1. The user checks or un-checks the desired options.
2. The Home key is pressed to return to the main screen that:
    - Makes the Options screen invisible and the Main screen visible.
    - Stores the state of the first check button in the **TinyDB** component.
    - Stores the state of the second check button in the **TinyDB** component.

In the **Screen1.Initialize** block (which runs whenever the app is restarted), the values for the two checkboxes are read in from the tiny database. And that's what persistent values means.

We can only have one TinyDB per app, but lots and lots of values can be stored in it; don't let the name fool you.

And this completes the Basic component drawer. For our next group of components, click on **Media** in the **Palette** column of the **Design** screen.

There is a checkbox just above the virtual phone screen in Design, shown in the next screenshot. Checking it makes all components visible (at least the ones that can be visible, not Clock and TinyDB, for example). I suggest you leave it off most of the time as it can really clutter the screen.

Media components

Left-clicking on **Media** reveals the five media-related components as shown in the following screenshot:

Here's a quick overview (with more detailed examples later on as we actually do things with them).

Camera

The **Camera** component takes a picture using your phone or other Android device's camera. After the picture is taken, the name of the file on the phone containing the picture is available as an argument to the **AfterPicture** event (a variable in **Blocks Editor**). An argument is a parameter that is passed to the component block and includes data. We then have the photo to do something with, such as setting it as the picture displayed by **Image** component. At the time of writing this book, App Inventor supports only rear-view camera.

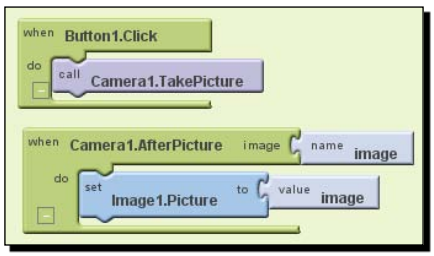
Hey, that might not sink in right away, so let me show you just how really easy using the **Camera** component is.

Time for action – shooting a photo

Okay, let's take a photo.

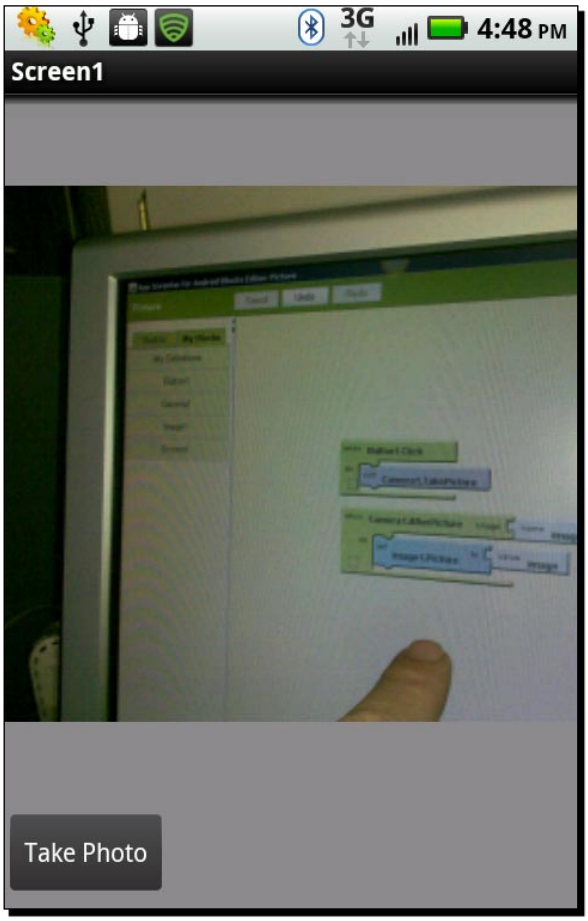
1. Drag-and-drop an **Image**, a **Button**, and a **Camera** component onto the virtual screen in that order. Change the text on the button to **Take Photo**.
2. Under **Properties**, make the **Image** (under **Width**) **Fill Parent**. Make **Height** `400` pixels.
3. Go to **Blocks Editor** and pull out **Button1.click** from **My Blocks**/**Button1**.
4. From **My Blocks**/**Camera1**, get **Camera1.TakePicture** and plug it into the button block.
5. Also from **My Blocks**/**Camera1**, pull out **Camera1.AfterPicture** (it will have a **name Image** block already attached, which will get the name of the photo we're going to take).
6. From **Image1**/**My Blocks**, get **set Image1.Picture to** and click it into **Camera1. AfterPicture** (see how App Inventor has taken your components from Design, created blocks, and filed them neatly away for your use!)
7. Now, from **My Blocks**/**My Definitions**, get value image and click it into **Image1.Picture**.

That's it, another complete app! It should look like the following screenshot:


when Button1.Click
do call Camera1.TakePicture
when Camera1.AfterPicture image name image
do set Image1.Picture to value image

Connect to your phone by using the **Connect to Device...** button at the top of the **Blocks Editor**.

On the Phone, press the **Take Photo** button, then take a photo as you normally do, then hit **Done**, and you should return to our little camera app, and there will be a photo in it now that should look something like the following:



What just happened?

We now know how to take photographs using App Inventor.

Yes, like almost all of App Inventor, using components and blocks to build apps is both powerful and requires minimal work.

ImagePicker

The **ImagePicker** component puts a specialized button in our apps, letting us view whatever photos are in the gallery on our phones and... wait for it... pick one of those images.

Want to see how it works? Use the little app we did just previously to take photographs. To do so, simply drag-and-drop an **ImagePicker** component onto the screen.

Now, go to **Blocks Editor**, leave the two blocks groups there as they are, and add the one following. You do so by dragging out an **ImagePicker1.AfterPicking** from **Blocks Editor**/**ImagePicker1**.

From **My Blocks**/**Image1**, pull out **Image1.Picture** and click it into place. Finally, in **Blocks Editor**/**ImagePicker1**, get **ImagePicker1.ImagePath** (which collects the path to your selected image) and click it in as shown in the following screenshot:

That's it. Click the button to show and select an image from your phone's gallery and the back button on your phone to return to your app. The selected image will be displayed in the Image component on your phone's screen. And, yes, it still takes photos also.

What you have now is the basis (needs some prettifying and enhancement) for a camera app that a) takes photos, and b) lets you see what you already have. This is something the standard camera app that comes with phones usually cannot do (that is, both shoot and review already saved shots)... Yes, App Inventor truly does give a lot of return for moving only a very few blocks.

Player

The **Player** component plays music and other sounds and so, too, does the **Sound** component that we'll examine next. The difference is that Player is meant to be a player—that is, play long sounds such as songs. **Player** provides the controls you expect a player to have—start, stop, and pause.

Using the player is so easy; I'll let you build your own right now.

Have a go hero – build a music player

Start a new project. Name it `player` if you like.

In **Design**, drag three **Button** components onto the virtual screen. Rename them **Start**, **Pause**, and **Stop**. Then, in the **Properties** column, change **Text** to match their names for each. It's good to get in the habit of naming components to avoid confusion later on.

Also, drag a **Player** onto the virtual screen (Player is an invisible component, so expect it to slide down to rest under the screen).

Use **Media** (bottom of the **Components** column) to upload a song or other sound file you want to play. The Player component will play MP3, M4A, OGG, 3GPP, and WAV formats.

For the time being, make sure that you do not pick one that is huge in size as AI and phones in general have trouble handling them.

Select **Player1** in the **Components** column, and in the **Properties** column, add your uploaded song as its source. Things should be looking as the following screenshot, and that finishes our player design.

In the **Blocks Editor**, pull out all three buttons' click frameworks from their respective drawers. From **Player1**'s drawer, get the start, pause, and stop blocks and click them in the right button framework. It will look as the following screenshot:

Connect to your phone and play your song. Pause it and tap the **Start** button on your phone; it'll resume playing at the stopping point.

Nice little player, not much programming. Later on, we'll jazz it up (pun, as ever, intended) by adding a way to select other songs and so forth.

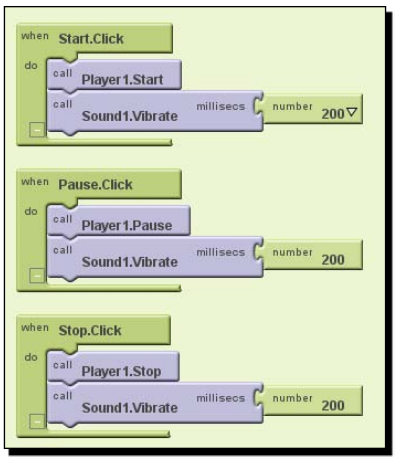
Sound

The **Sound** component is designed for sound effects and other short sounds. It's great for adding all the bleeps and bloops and so on. You upload a sound effect to **Media** and attach it the same way as we did in **Player** previously.

Using the player app you did previously, let's make the buttons vibrate when we touch them (so we know we got a good hit on it).

Time for action – vibrating buttons

1. In **Design**, drag a Sound component (another invisible one) and drop it on the screen. That all's we do there; nothing to attach for this purpose.
2. In **Blocks Editor**, open the **Sound1** drawer and drag out **Sound1.Vibrate**. Click it into the **Start** button. Get a numerical variable (**Built-In**/**Math**, top choice), click it in, and change it to `200` (that's 200 milliseconds or about 1/5 of a second).
3. Click on **Sound1.Vibrate** to select it. Type *Ctrl + C* to select, click anywhere blank on the **Blocks Editor** screen, and type *Ctrl + V* to drop the copy (the number variable will be included). Drag it into the **Pause** button framework.
4. Click on a blank area again and do *Ctrl + V* to drop a second copy. Drag it into the stop button framework. Your blocks now look like the following screenshot:



Check it out on your phone. All three buttons now vibrate when tapped. Fun, yes?

VideoPlayer

The **VideoPlayer** component ... well... it plays videos. The component works quite similar to the Player component except, of course, it will play video (with sound). Formats include 3GP, MP4, and (App Inventor being developed by Google) Google's new WebM format.

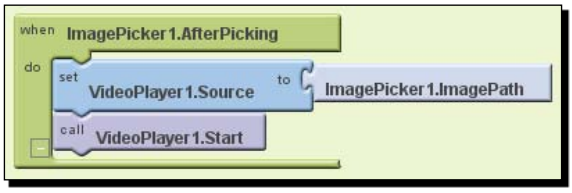
There are limitations, especially in the size of your videos (and other media files).

The App Inventor help text for **VideoPlayer** states: "App Inventor for Android only permits video files under 1 MB and limits the total size of an application to 5 MB, not all of which is available for media (video, audio, and sound) files. If your media files are too large, you may get errors when packaging or installing your application, in which case you should reduce the number of media files or their sizes ..."

Most movie editing software can help you reduce the size of video files you might want to include in apps. We'll do some more powerful video apps later in the book (I love videos!), but here's all you need to play video right now:

Drop our old friend the **ImagePicker** component and a **VideoPlayer** component onto the virtual screen.

In Blocks Editor, below is all you need for a video player. Controls to fast forward, rewind, and stop will be automatically included.



And that completes the **Media** drawer of the **Palette** column. **Animation** comes next.

Animation components

There are only two animation components currently available—**Ball** and **ImageSprite**.

Ball

We met the **Ball** component earlier in this chapter when exploring the **Canvas** component. We use **Canvas** for drawing and to do animations on and to control the area where they happen.

Balls and **ImageSprites** are ideal for use in arcade-style games, for example, because they can react to touches and to collisions with each other and with the edges of the **Canvas**.

I guarantee you we'll have fun with both these elements, especially in the chapter on games.

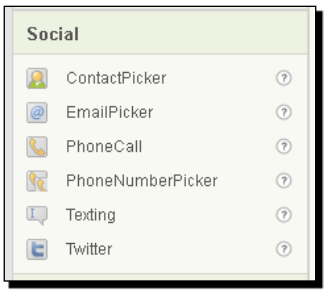
ImageSprite

The **ImageSpite** component works exactly as the Ball component except we use an image instead of an automatically generated ball graphic. I like using PNG to get the transparent background, like in the game I wrote below, but more about all that as we progress through the book.

Social components

Social components involve interaction (communication) with other people. Phone calls, e-mails, Twitter tweets, Facebook, and so on are all social activities.

The following are the **Social** components currently available in App Inventor:



Here's a brief introduction, but we will become real buddies with them in quite a few examples later.

ContactPicker

The **ContactPicker** component is a specialized button (such as **ImagePicker**). It displays a list of contacts to choose from among those on the user's phone. After a selection, we can have the app set various properties for displaying the chosen contact such as:

- **ContactName**: the contact's name
- **EmailAddress**: the contact's primary e-mail address
- **Picture**: the name of the file containing the contact's image, which can be used as a **Picture** property value for the Image or **ImageSprite** component
- Other properties affect the appearance of the button (**TextAlignment**, **BackgroundColor**, and so on) and whether it can be clicked on (enabled)

EmailPicker

The **EmailPicker** component provides a specialized textbox that lets the user enter an e-mail address from their phone's contact list.

E-mail pickers are usually used with a button. The button is tapped when the e-mail address is complete. I'll show you auto-completion and other ways of using this component in the upcoming chapter on writing communications apps.

Download from Wow! eBook <www.wowebook.com>

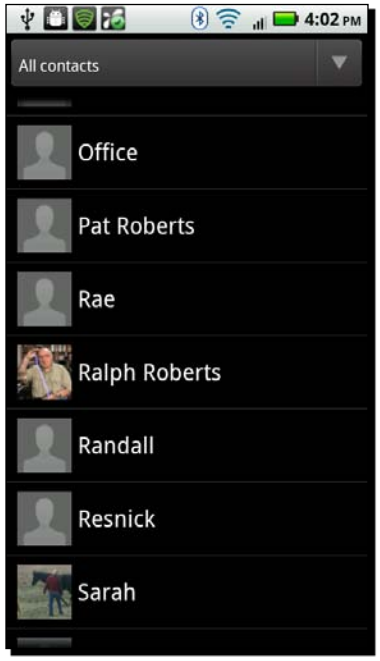
PhoneCall

The **PhoneCall** component is a non-visible component that makes a phone call to the number specified in the **PhoneNumber** property. We will use this in conjunction with both the **ContactPicker** and the next component, the **PhoneNumberPicker**.

PhoneNumberPicker

Well, this component picks a phone number, eh? To be more precise, it provides a specialized button that when tapped, opens up the phone's contact list.

There's no configuration needed for the basic action. Just drop the component on the virtual screen and you'll get the button. Connect to your phone through the Blocks Editor and tap on the button. You'll have your contact list open as mine, shown in the following screenshot:



Of course, there's more to it than just that, and we'll exploit those capabilities in apps later.

Texting

We used the **Texting** component in our `NoTextingWhileDriving` app. We'll delve into all aspects of texting and how App Inventor can manipulate texting functions by example, especially in the communications chapter (that's *Chapter 5*, *Apps That Communicate*).

Twitter

I love tweeting... follow me at `http://twitter.com/ralphr`. And the Twitter component gives us access to:

- Setting the status of the logged-in user
- Directing a message to a specific user
- Receiving the most recent messages directed to the logged-in user
- Following a specific user
- Ceasing to follow a specific user
- Getting a list of users following the logged-in user
- Getting the most recent messages of users followed by the logged-in user
- Getting the most recent mentions of the logged-in user

You must obtain a Consumer Key and Consumer Secret for Twitter authorization specific to your app from `http://twitter.com/oauth_clients/new`.

We will have fun with Twitter apps.

That's it for introducing the more basic functions.

Sensor components

The sensing of acceleration (how fast the phone is moving), location (the actual coordinates on the Earth), and orientation (how you are holding you phone) get handled by the three components in this **Palette** drawer. We'll master them in *Chapter 4*, *Mastering Concepts and Advanced Components*.

Screen arrangement components

Vertical, horizontal, and table screen arrangements are critical in designing your app so that it is both neat and readable. We'll devote a proper amount of time and space to this in *Chapter 4*, *Mastering Concepts and Advanced Components*, but also you'll get to play with them in the next chapter—and every other chapter in the book too.

With the limited screen real estate on an Android phone, making your app's display readable is critical. Screen arrangements ease this task dramatically.

LEGO® MINDSTORMS® components

Oh, these components are downright cool! They let you run robots! I'm serious. Lego Mindstorms NXT is a programmable robotics kit with all sorts of options. They are not especially cheap, but they are fun.

App Inventor includes these components allowing us to write apps and control these robots with our Android devices (seems appropriate to use my Droid 2 that way). More on talking to robots in *Chapter 4*, *Mastering Concepts and Advanced Components*.

Other stuff

There are more powerful components left—GPS, Bluetooth, Text to Speech, TinyWebDB (database access over the Internet), Google Fusion Tables, and so on. Make a date with *Chapter 4*, *Mastering Concepts and Advanced Components* for those.

Not ready for prime time

This drawer includes components still in development but with useful operations. We look at these in detail later in this book.

Summary

In this chapter, we explored the use of Google App Inventor's web interface (called **Designer**). We then looked at the various sets of components — basic, media, animation, social, sensor, screen arrangements, LEGO® MINDSTORMS® components, other components, and the not ready for prime time components.

Using our introduction to components in this chapter, we are now ready to bring up the part of App Inventor on our local computers (**Blocks Editor**) and make the components do our bidding by simply playing with blocks. You got to love it!

3
Playing with Blocks

When I was a little boy, I loved playing with alphabet blocks. Bet you did also. Well, App Inventor brings that joy back—we get to play with blocks again!

Blocks of code, but we never have to see the code, let us tell the components what we want them to accomplish. It's all visual now, dragging-and-dropping blocks on our computer screen into various simple configurations.

We're actually programming (shhh! Don't tell anyone!). It doesn't look like programming, and you certainly don't have to know much about programming technicalities to create powerful and complex programs (that is, in operations and not in the building of them).

Blocks are simply the logic that instructs the component services how to act and in what order tasks happen.

In this chapter, we look at the types of blocks and how they are used, and find our way around in the App Inventor Blocks Editor.

In this chapter, specifically, we will learn:

- The Blocks Editor
- Definition blocks
- Text blocks
- List blocks
- Math blocks
- Logic blocks
- Control blocks
- Color blocks
- Individual Component Blocks

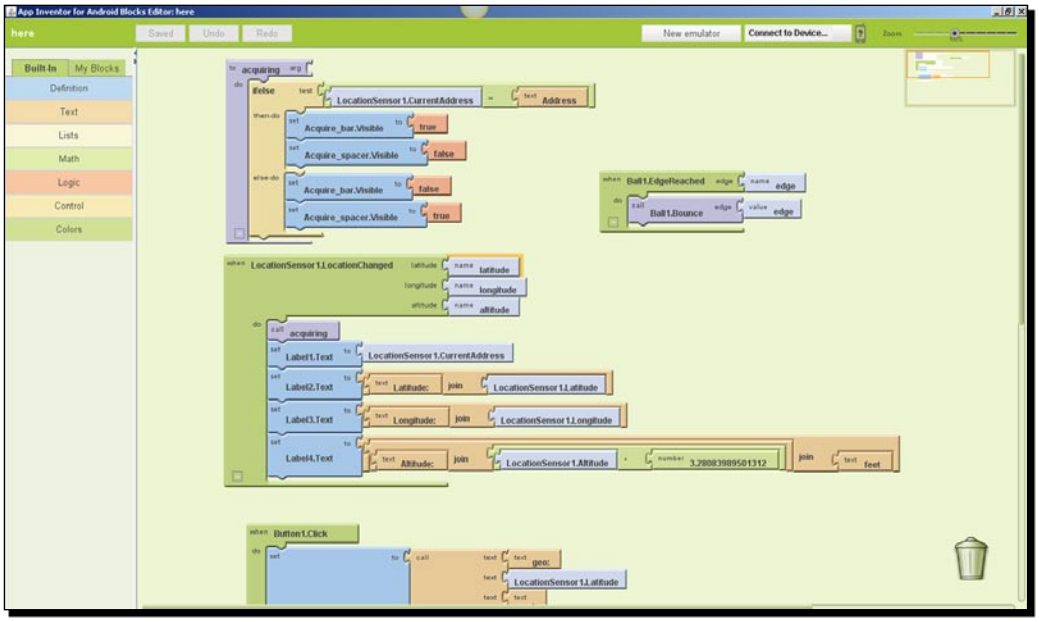
We have, of course, already used blocks in the preceding two chapters and will use it throughout the rest of the book. But, this chapter allows us to explore the mechanics and details of blocks, laying a foundation (so to speak) that lets us build more complex apps in the coming chapters.

First, let's get more familiar with the operation of the **Blocks Editor**.

The Blocks Editor

The Google App Inventor **Blocks Editor** is the part of App Inventor that runs on our local computers. As we've done several times already in the previous two chapters, we start **Blocks Editor** by clicking on the **Open the Blocks Editor** button at the top-right side of the online **Design** screen.

Blocks Editor, a Java application, is downloaded from the web each time we wish to use it. This has the major advantage of ensuring we are always using the very latest version. The following is the Blocks Editor on my computer (a Windows 7 machine) with the blocks of an app in progress (this will be an example in a later chapter, shaping up very nicely).



The top bar

No, not that kind of bar—it's the green one along the top of the **Blocks Editor** screen. Look at the left side first (see the following illustration). The first word(s) on the left of the green bar is the title of the app. Next, we have three buttons.

The first button (leftmost) currently reads **Saved**, meaning the app has been saved out on the cloud to the App Inventor site (that is, in your **My Projects** area).

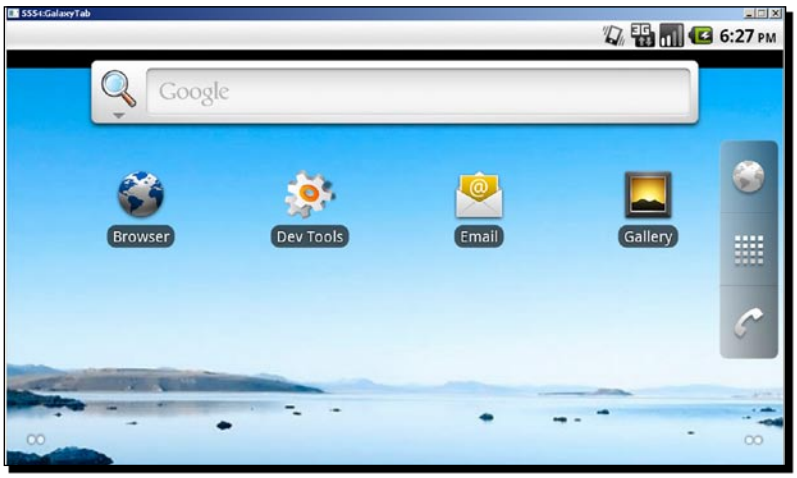
The next two buttons—**Undo** and **Redo**—steps you back or forward in the history of changes we've made to our block groups. Just click on **Undo** once or more to step backwards and **Redo** once or more to (got to say this) undo the **Undo**. That could be a song title, eh?

These two handy buttons can save you much work; they let us back gracefully out of blunders or restore it if we were right the first time.

On the right side of the green bar, shown in the next screenshot, are four items, the first being **New Emulator**, which allows us to start up a virtual device. Remember from *Chapter 1*, *Obtaining and Installing Google App Inventor*, we found this takes quite a bit of time. Again, I prefer to use my phone connected via USB cable to my computer for testing purposes. It's faster and truer to how the final app will look and run.

When do we use the emulator? If we don't have a phone or, say, you want to do apps for tablet computers but you don't have one yet. For example, below is an emulator I just found and installed for Samsung's Galaxy tablet.

While we cover apps for tablets briefly in this book, mostly we concentrate on phone apps.



Emulators are installed in the Android SDK, which is beyond the scope of the book, except I'll mention it several times as an enhancement to App Inventor.

If you do not have the Android SDK (Software Development Kit) on your computer, it is recommended you add it—there are several ways in which the SDK enhances creating apps with App Inventor. Instructions for installation (it's free) are at: `http://developer.android.com/sdk/installing.html`.

One useful function of the Android SDK is to get screen captures from your phone.

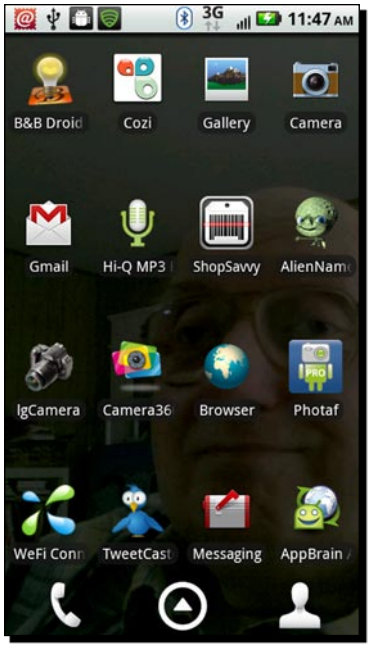
Have a go hero – capture screens from your phone

1. On your computer, find the `android-sdk` directory in the `tools` subdirectory.
2. With your phone attached by USB cable to your computer, double-click on `ddms` (`ddms.bat` on Windows)—the Dalvik Debug Monitor.
3. Click on your device's name (it will be the top one).
4. Click on **Device** (top menu line), then on **Screen Capture**.

You'll get a gorgeous capture of your phone's screen. Save it to your computer if you like (saves as a PNG file).

If the file already exists, it will be overwritten without any alert or confirmation.

The following is my Droid 2's current main screen (I love apps, these are the ones I use most).



Now, back to Google App Inventor and the right side of the top green bar—left-clicking on the **New emulator** button starts up the default emulator.

Left-clicking on the **Connect to Device** button (in the following case) gives us the choice of connecting to our phone (that's the serial number) or an emulator if one is active (the one shown is the one for the Samsung Galaxy pictured earlier).

The preceding small phone icon shows if you have a connection or not. When connected, it looks like this (question mark changes to an arrow):

Finally, on the extreme right of the green bar and just below it, we have some handy visual and navigation features as shown in the next screenshot. Once we have sophisticated apps, there'll be blocks all over the screen, and you'll be thankful to have some space (especially since the latest release of App Inventor just dramatically increased possible vertical space).

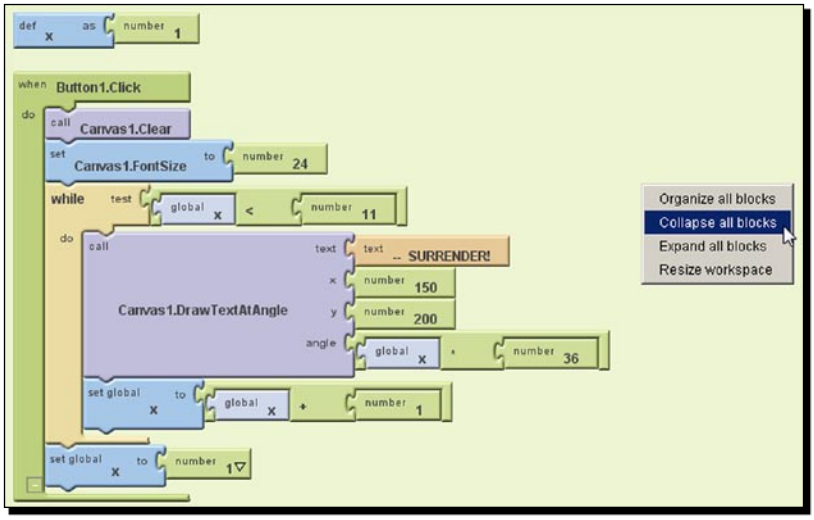
Zoom lets us to resize the view larger or smaller. Just click on it to grab the slider and (holding down the left mouse button) move it left to make smaller or right to increase the size of the blocks in the current view.

Below the green bar, left-click in the red box, hold down the left mouse key, and move it around to change the view window. The portion of the overall **Blocks Editor** screen shown on your computer screen matches what the red box shows in smaller scale.

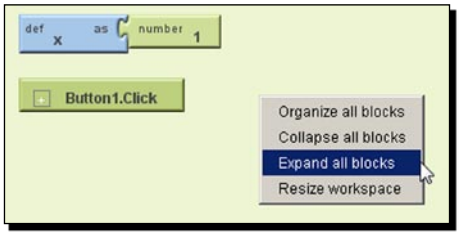
Now, moving down to the main screen (the cream-colored large area), a couple more features make our lives easier as we create apps.

Time for action – collapse and expand block groups

First, right-click anywhere on the main area of the **Blocks Editor**. A small four-item menu pops up as shown in the following screenshot that will help you to manage blocks.



Left-clicking on **Collapse all blocks** as shown previously causes all the block groups to condense, as we see in the next illustration. Clicking on **Expand all blocks** brings them back out where you see all component blocks of every group.



Clicking on the small + (plus sign) on a collapsed group expands only that group. Clicking on the—(minus sign) causes only that group (when expanded) to collapse.

When we make changes in the arrangement of blocks, these changes are automatically saved and will be in the new configuration when Blocks Editor is opened again.

The other two choices on the small menu we get, again, by right-clicking anywhere on the main **Blocks Editor** screen, are also useful.

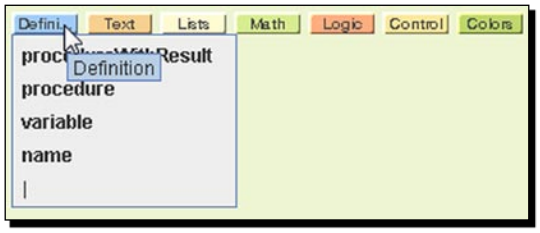
Organize all blocks moves and aligns all block groups into the smallest, neatest area possible.

If blocks get lost off from the screen, you can click **Collapse all blocks** and then organize again to get the blocks back onto the screen.

Resize workspace, as the name suggests, makes the workspace bigger. To test it for yourself, move a block slightly offscreen on the right or bottom edge. Click on **Resize workspace**, and you'll see the orange block inside the green box in the upper-right of the screen jump slightly in size. Scroll your screen to the right or down to see the increased workspace area.

Left-click anywhere in the main area and hold down the left mouse key, and then you can move the view area about to see all of your block groups.

Finally (the second of the two items I mentioned previously), if you left-click anywhere in the main area and do not hold down the mouse key, the shortcut bar shown below pops up. It's a quickie way of adding **Built-In** blocks without having to open the drawers in the left column of the **Blocks Editor** screen (which we will be exploring in the next section). As you become more experienced with the Blocks Editor, you'll find yourself using these shortcuts more and more.



What just happened?

We now know how to expand and collapse block groups for our convenience in keeping track and making sense of the blocks for the apps we create.

But, what if we want to delete blocks?

Time for action – deleting unwanted blocks

Finally, let's cover deleting unwanted blocks. Pull a block onto the working area of the **Blocks Editor**—does not matter which one, any one will do.

There are two ways of deleting unwanted blocks. First, move the mouse cursor over the block and hold down the left button. Drag the block over the small garbage can on the lower-right of the **Blocks Editor** screen. When the lid opens, release the button. The block is trashed.

For the second method, left-click on the block to select it. An orange outline appears around it showing active selection. Hit the *Delete* key on your keyboard, and the **Please choose...** dialog box appears. Click on **Yes** to delete.

Deleting any block with other blocks plugged into it trashes all the blocks in that group.

Now, it's time to start playing with blocks seriously. Well, kind of seriously; it's truly a lot of fun to program this way.

In the rest of this chapter, we will be mostly covering the **Built-In** blocks, but with examples that show component blocks—those under the **My Blocks** tab—in action too.

What just happened?

We now see how easy it is to safely delete blocks.

As a convention, we normally refer to blocks by the large text on them, as in **Button1.Click**. However, note the smaller words on the block as they tell you what action it performs, as in **when Button1.Click do**.

Definition blocks

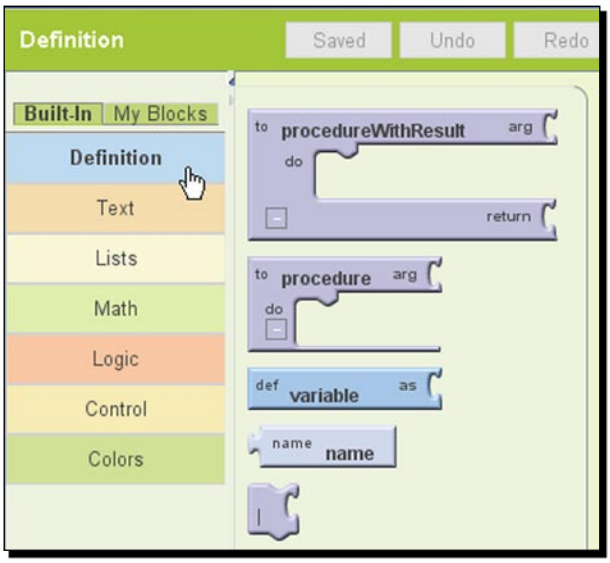
On the left side of the **Blocks Editor** screen, we have two columns of blocks. Left-click on the column tab—**Built-In** or **My Blocks**—to reveal the headings (we call them "drawers" in each of those two categories (see the following screenshot for location):

Starting a new app in **Designer** then opening the **Blocks Editor** and clicking on the **My Blocks** tab shows us that there is very little in it initially. The **My Definitions** drawer is empty because we have not defined anything yet (but will in this section). The **Screen1** has a number of blocks relating to screen control in it because the **Screen1** component exists by default. Currently, only one **Screen** component per app is possible.

One quick note about the statement, "only one Screen component per app is possible". Those of you with a little programming experience are already screaming (and rightly so) "you can't do anything with just one screen!" Well, not to worry—we can have as many virtual screens as we like, and I'll show you how over the course of numerous example apps in this book. Because it's true: you can't do anything with just one screen.

Back to blocks—there's little to see yet in the **My Blocks** area. It gets populated automatically by drawers full of blocks as we drag components onto the phone screen in Apps Inventor **Designer**, and we'll pick those up as we go along.

Click on **Built-In** (see the following screenshot for the list of drawers that gives us). These drawers of blocks are always full of the same blocks. We use them to complement, enhance, and control the blocks that appear in the **My Blocks** area. The **Definition** drawer is open in this screenshot:



There are blocks giving us text and mathematical manipulations, structuring of lists, logical operations (true or false... `true` in this case), control blocks for loops, color designation, and so forth. In other words, this area has seven drawers of utility blocks (currently, at least. Always remember that App Inventor is under development with new changes and additions coming regularly).

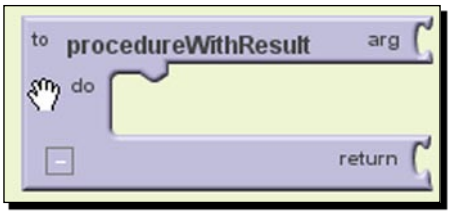
For the remainder of this chapter, we'll examine each of these drawers, beginning with the **Definition** drawer (refer back to the previous screenshot to see its location).

procedureWithResult

Those of you who have programmed would know the term subroutine, procedure, method, or function. That's where we write a block of often used code and use it over and over. We name it and just call it by that name whenever we need it. This really speeds up programming and is the way (of two possible) we'll most often use this block. In short: *write once*, *use many*.

To move a **procedureWithResult** block into the **Blocks Editor** working area, just left-click on **Definition** in the **Built-In** column (see preceding illustration) and, while holding down the left mouse button, drag out the block. Here's what the **procedureWithResult** block looks like:

Download from Wow! eBook <www.wowebook.com>



The name is lowercase instead of the convention we've been using in capitalizing names of App Inventor components. The reason for this is that the *procedure* on the block is simply a placeholder. We change the name of the block to fit our needs. Examples follow shortly.

Procedures, as I said, may be used two ways. In addition to writing subroutines, a **procedureWithResult** block may also be a *function*.

Yeah, another bit of programmer jargon. While App Inventor is designed for use by non-programmers and doesn't look like programming (since we drag blocks instead of typing long mystifying lines of code), the programming concepts are still there underneath the blocks. Knowing some of it helps us create better apps (or at least ones which actually work). So, let me define *subroutine* and *function*:

1. **Subroutine**: A subroutine in programming is a named group of lines of code used more than once, thus saving time (type once) and space (use many). In App Inventor, a procedure is a named collection of blocks created once and used by invoking it whenever needed.
2. **Function**: A function is like a subroutine (named group of lines of code), except we can pass *arguments* (input numbers or other data) to it and get *results* (answers based on the input). In App Inventor, this is accomplished by plugging a certain type of block (*name* block) into the *arg* socket on the **procedure** block (and there can be more than one argument).

In short, subroutines save the drudgery of writing the same collection of blocks over and over, while functions do more specific calculations or other manipulations for you. An example of the latter might be trimming trailing spaces off from any string you send to it.

Let's start with an example of a function.

Calculating the area of a circle is done with the formula $\texttt{pi} \times \texttt{r}^2$ where $\texttt{pi}$ (a mathematical constant) = 3.14159... and $\texttt{r}^2$ = the radius (distance from the center of the circle to its edge) times itself (squared). So, let's do a quick app that finds the area of circles.

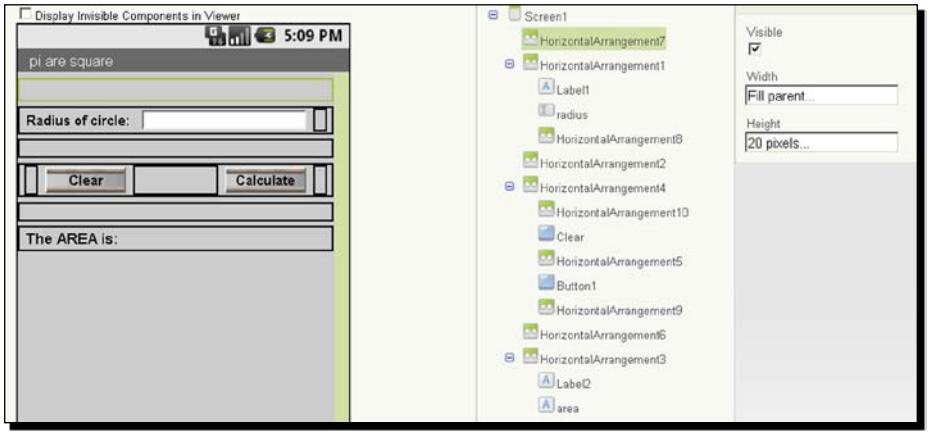
We normally learn how to find areas of circles in elementary school. When I was in school, the teacher would say, "the area of a circle is *pi r squared*", and some funny kid (okay, usually me) up here in the Carolina mountains would pipe up, "no ma'am, cornbread are square, pie are round."

Anyway...

The following is an example of using the **procedureWithResult** block. We'll spend a bit of time on it and make it more involved than most examples in this book because I'll also be emphasizing design and layout techniques (making your apps look good) as well. It pays to expend a bit of extra effort, and App Inventor gives you the tools to neaten up the appearance of our apps. Let's build it.

Time for action – calculating the area of a circle using a function Design

1. First, start a project in **My Projects** (I named mine `pi_are_square`). Designer opens it for you on the **Design** screen. Start the **Blocks Editor** and then (in the **Blocks Editor**) connect to your test device, which is needed to show the appearance of your app in real time as you design it.
2. Return to the **Design** screen in your web browser.
3. In App Inventor **Designer**, we need a label and a **Textbox** component (for entry of the radius), a button to clear the previous entry, and two labels to show where the answer appears and the calculated area.
4. One very good habit to get into, even on simple learning apps like this one, is taking the time to format it. Turning out apps that both look and work professionally should be our end goal, and this is quite easy in App Inventor.
5. Use the **HorizontalArrangement** and **VerticalArrangement** screen arrangement components we met in *Chapter 2*, *Learning Components* for spacing and centering (as in the following screenshot, taken from the **Designer** screen). Note the following *outline* appearance: it means that the indented components are *inside* the outdented ones.



In App Inventor **Designer** (as we learned in *Chapter 2*, *Learning Components*), we select the components needed to enable our app to perform its function(s) and create the cosmetic look (the way the app appears on the end user's phone).

In designing an app, always think of some stranger who knows nothing about you, App Inventor, programming, or much of anything else. Create apps that 1) this hypothetical dummy cannot mess up and 2) look good to him or her. Then, you have an app that could take off and be popular with hundreds or thousands of people—because most people are smart enough to appreciate your extra work, whether they understand it or not. And, they will know if it's not there.

Okay, referring back to the previous **Designer** screenshot, here are some of the things I did to make the app both functional and look good (and check out the final look in the two following illustrations, captured from my phone being used as a test device). Try these for yourself. They're worth the effort, because doing it this way a few times makes designing awesome-looking apps second nature.

6. Put a **Horizontal Screen Arrangement** at the top of the screen (drag-and-drop it from the **Screen Arrangement** drawer in the **Palette** column). With it selected, it turns green as shown in the previous screenshot, in the **Properties** column set the width to **Fill Parent...** (as wide as the screen) and the height to `20` pixels. This gives you a top margin (think of a page. People have been creating documents with margins for thousands of years—it looks good and is what folks subconsciously expect to see).

7. Drag another horizontal arrangement beneath the first. Set width to fill the parent, and leave the height to **Automatic** (the default value). This will be the container for our parameter entry (the radius of the circle to calculate the area for).

8. We need a **Label** (drag it into the green horizontal arrangement box), and a **TextBox** to enter the radius. Drag in the latter, place it to the right of the label in the horizontal arrangement, and name it **radius** (use the **Rename** button at the bottom of the **Components** column while the textbox component is selected).

9. In the **Properties** column for each (click on the component to select it), change what you like to fit your app (in this case, set to numbers only). Change the **Text** field for the label to `Radius of the circle:` (I put a space in front of the text for a margin and made the font larger and bolded it. Do whatever looks good to you). In the textbox's properties, just make the text blank—the user will enter numbers here. In the **Hint** field, put `enter radius here`. Add another horizontal arrangement to the far right—`10` pixels wide and **Fill Parent…** for a right margin.

10. Now, below the horizontal arrangement with the preceding two items, put another as a spacer. Make it **Fill Parent…** and `15` pixels high (or, again, whatever looks good to you; you're designing the appearance of this app).

11. Now, another horizontal arrangement as a spacer—set height to `15` pixels.

12. Another horizontal arrangement below that—set fill as **Automatic** (the default value).

13. Drag in yet another horizontal arrangement into the one above for a left margin. Make it `10` pixels wide and **Fill Parent…** for height.

14. Now, pull a **Button** component beside the margin spacer (make it **Fill Parent…** and leave height to **Automatic**, and rename the component to `Clear` to avoid confusion in the **Blocks Editor** phase; you also use the convention of naming all buttons, for example as `btn_Clear`, so that all components of that type are grouped together for easy finding), another horizontal arrangement in the center (fill, fill), another button to fill, auto, both label (change the **Text** box and rename by clicking on the component in the **Components** column and using the **Rename** button at the bottom of that same column) to `Calculate`, and a final horizontal arrangement for the right margin (`10` pixels wide, fill).

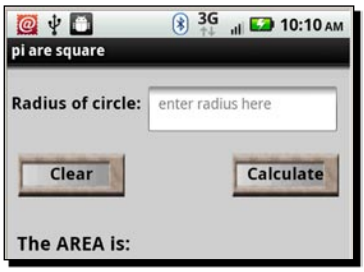
A powerful but easy way of fancying up your apps is by using images for buttons (like I did, see the next screenshot). You can make your own button in software such as Photoshop or Gimp or simply find one of the zillions of web button designs marked free for reuse on the Internet. Download one you like to your computer and upload **Design** (at the bottom of the **Components** column, click on **Add** in media and choose the button file to upload). Then, click on the button component; from properties, click in the **Image** field, select the button image from the drop-down list, then click **OK**. That's it.

15. Another horizontal arrangement. Make it **Fill Parent…**. Set height `10` pixels.

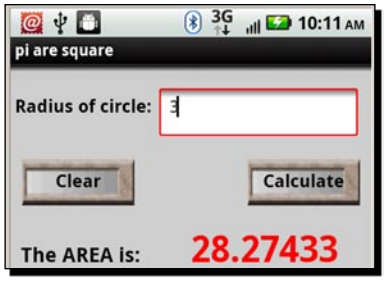
16. Below that, place yet another horizontal arrangement. In this one, put two label components—one is `The AREA is:` and the other blank for where the calculated results will appear.

While you're doing all of the above, just glance over at your phone and the results appear in real time. Please remember, **Design** is not what you see, it's what you get, and it's an approximation. So, we really need a test device to see what it really looks like on a phone or other device. And, later in the playing with blocks to build the app logic phase, test to make sure our blocks are working right.

Here is the appearance of my design effort. Note, the screen title is set in **Design** by clicking on the top component (the screen) and typing it in the **Properties** column.



I like it. And, skipping ahead just for a moment, here is what it looks like in a finished, operating app:



What just happened?

We just completed an example exercise in which we learned how to create the design for creating polished and professional-looking apps by just using a few **HorizontalArrangement** components. Throughout this book, I'll present you with more design elements and techniques that will make both your apps and you look good.

Now, let's put together the blocks, thus enabling our app to calculate.

Time for action – calculating the area of a circle using a function

Switch over to the **Blocks Editor**.

First, we build the function procedure to calculate the area of a circle.

1. Under **Built-In**/**Definitions**, pull out a variable (we need a place to store the calculated area). Left-click on the word **variable** to edit and rename it as **area**. From **Built-In**/**Math**, get a **number** block (the top one) and plug it into your **area** variable. This defines the variable as one holding numbers. If we'd pulled out a **text** variable from **Built-In**/**Text**, we'd have a text—a variable that expected to hold letters. You can leave the number variable at its default of `123`, but it's best to set it to a value, especially if you're using the variable as a constant. But, generally, it's *type* not *value* that matters in defining a global variable. The defined variable looks like this (and can be anywhere on the screen):

.12

Key concept of how App Inventor works:

The global variables (work everywhere in the app) and procedures that you pull out from the **Built-In**/**Definitions** drawer automatically appear—ready for use—in the **My Blocks**/**My Definitions** drawer, just as components you drop in on the **Design** screen also create drawers of block goodies that relate to them in the **My Blocks** tab. Open the drawers and drag out blocks as you need them. You can use them over and over, renaming as needed.

2. Also from the built-in definitions drawer, drag out a **procedureWithResult** container block. Rename it to `area_of_circle` (use underscores for spaces or you'll get an error message).

3. From the same drawer, we need to get and define (in this case by just renaming) a couple of **name** variables. Name variables differ from global variables, where their use is localized (in this case, in our **area_of_circle** procedure rather than working anywhere as the *global* variables do). Edit your two name variables to `pi` and `radius`.

4. Now, click them into the **arg** (argument) sockets on the **area_of_circle** procedure container (as you click in each one, another empty **arg** socket is created, so you never run out).

5. Next, we use our global variable to collect the calculated area answer within. From **My Blocks**/**My Definitions**, drag out the **set global area to** block (shown in the next screenshot). Also, from **Built-In**/**Math**, grab a couple of multiply frames as shown in the next screenshot with the **x** or *times* sign on them.

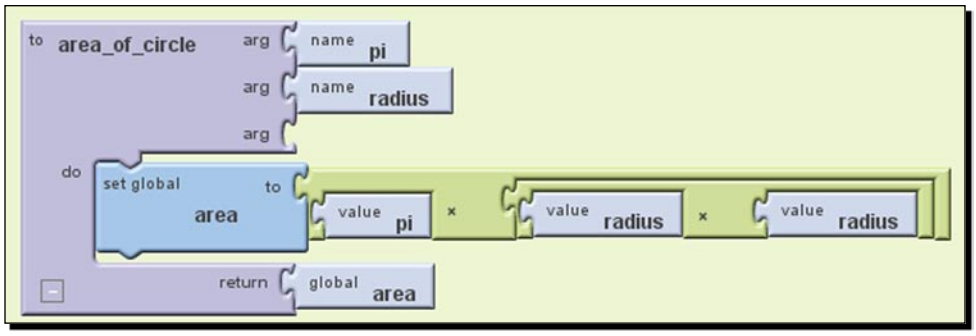
6. Click the **set global area to** block into the **do** socket in our procedural function container.

7. Put one of the multiply frameworks into the other as shown, and click them into the **area** global variable.

8. From the **My Blocks**/**My Definitions** drawer, get the **value pi** (generated when you created the **name pi** block) and the **value radius** blocks and plug in as shown in the next screenshot. Since the formula is squaring pi (multiplying it by itself), we need another **value radius**. Either slide one out of your **My Definitions** drawer or simply click the one on the screen to select it, then do *Ctrl + C* (copy) and *Ctrl + V* (paste), and drag the copy into place.

9. One final thing. Go to **My Definitions** and get a **global area** block (also created automatically when you defined the global variable for area when we first started). Click it into the **results** socket at the bottom of the **procedureWithResult** block container.

The following is the completed function to determine the area of a circle:



What just happened?

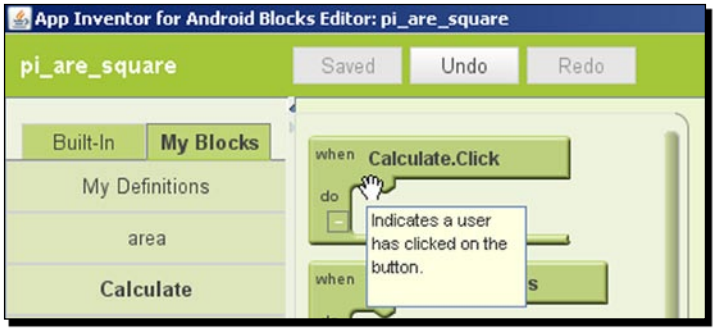
And now we have a *function* (or *procedure* in App Inventor terminology) that—when called in an App Inventor app—uses the two arguments (`pi` and the `radius`) in the formula *pi* times (radius times radius) to return the area of a circle.

Now, we'll see how to feed it arguments, call it, and use the result.

Time for action – finishing the "pi are square" app

The rest is simple. We simply set up the two button components we added in **Design** to control our app.

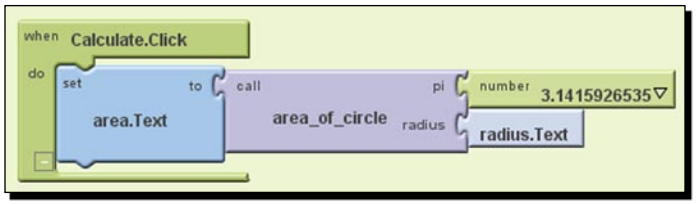
Did you rename the first one **Calculate**? If so, we can find it easily in **Blocks Editor** by looking in the **My Blocks** tab and clicking on its name—that opens the drawer of blocks created for you when this button component was dropped onto the virtual phone in **Design**. Slip out the **Calculate.Click** framework block, which looks as shown in the following screenshot:



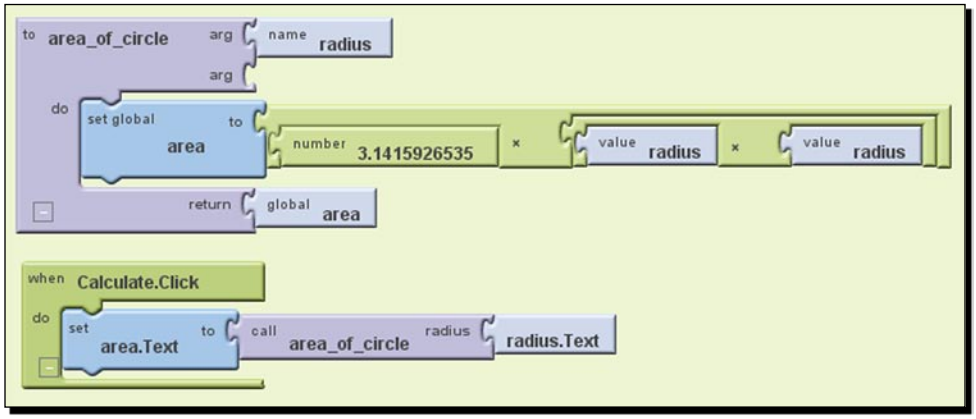
When the end user presses our **Calculate** button, the app sends the two arguments (`pi` and `radius`) to the function we just built, and the area of the circle defined by the radius (distance from center to edge of the circle) is returned and displayed.

The following is how it looks after we complete the button, and here is how we build it.

1. In **My Blocks**, pull **set area.Text to** from the **area** drawer created automatically from the label component we renamed **area** in **Design**. Snap it into the button (don't you just love that little click?).
2. From **My Definitions**, we get a **call area_of_circle** block and click it into **area.Text**.
3. We add the two arguments to be passed to the function for calculation—*pi* (I got it to 10 places off the Internet—more than accurate enough—and put it in a **number** block snagged out of **Built-In**/**Math**) and pull out **radius.Text** from the **radius** drawer as shown in the following screenshot (again, created when we renamed the **TextBox** component **radius** in **Design**).



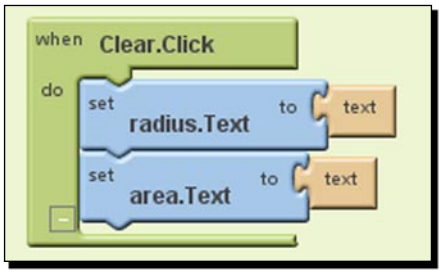
In the previous screenshot, I'm passing **pi** as an argument, which is unnecessary. It's really a constant used in every calculation. So, you cannot add it above and just replace the name variable in the area calculation function with a **number** block from the **Built-In**/**Math** drawer and put `pi` in that, like this:



Simplifies it a bit—even I can be improved on.

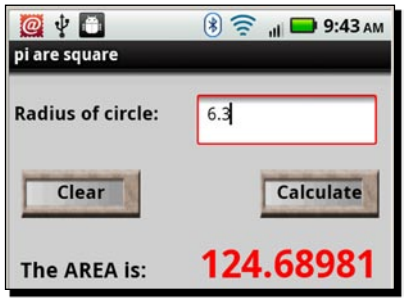
The other button we defined is **Clear**, and it's for the convenience of your users to clear the textbox and result label (**area**), making ready for the next area to be calculated.

Pull out the **Clear.Click** framework, and just set the textbox for *radius* entry and the result to a blank by using two **text** blocks from the **Built-In**/**Text** drawer (it's the top one) and editing it to remove the default value (click on the second **Text** in the block and delete it).



What just happened?

We now have a complete app for calculating the area of a circle when we know the radius.



One final programming concept:

After your app (any app) is working, test it for *bugs*. We've already defined this term as 'mistakes' in programming. Well, mistakes can be omissions (*oops, let that out!*), too.

In our app here, there is an omission that can cause the app to bomb, crash, and burn (lots of ways of expressing it) or stop working with an error message. This is not good.

Try this. Enter a letter instead of a numbers for your radius. Yep, it did not like that! We get an error message like so:

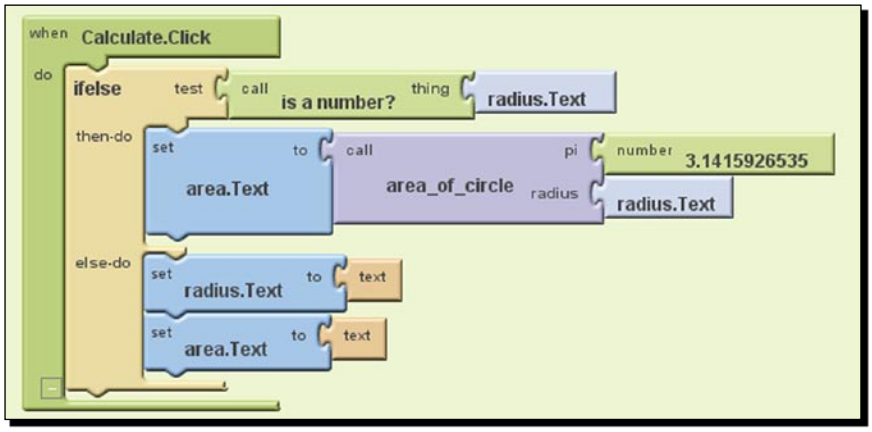
An immutable law of the universe is that if an end user can mess up, they will. And, who gets blamed when it happens? That's right, us—the app creators.

So, good programming practice is to include error traps—code that senses a user boo-boo and allows a graceful recovery without trashing the program's operation. The same applies in App Inventor.

Time for action – adding an error trap

We can add an error trap against letters in the **radius** textbox crashing the app by a slight addition to the **Calculate** button.

1. From **Built-In**/**Control**, latch onto an **ifelse** framework and pull it out.
2. Detach **area.Text** from the click button frame (the blocks plugged into it will follow along), and click this assembly into the **then-do** socket of the **ifelse** framework.
3. Click the **ifelse** (and all its attached blocks) into the **Calculate.Click** button frame.
4. The **ifelse** needs a test so it knows what to do or not do. Easy to set up. In **Built-In**/**Math**, grab (**is a number?**) and click it into the **test** socket.
5. Click **radius.Text** into the **is a number?** socket.
6. Finally, we give this something else to do by copying the two clear operations from the Clear button click frame, and we are done (see below for how it looks). Our test is now complete.

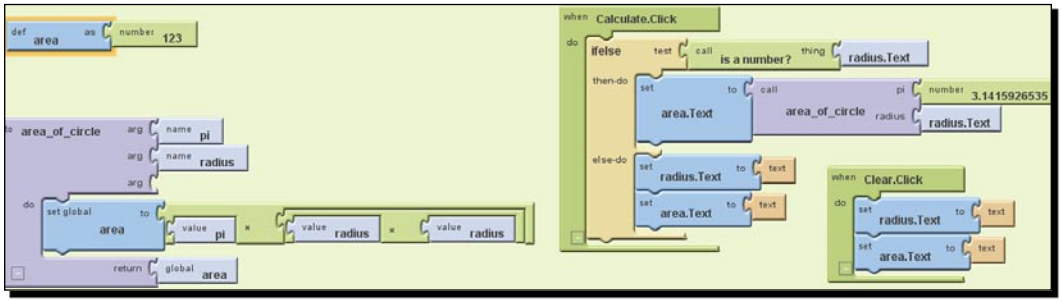


An alternative is to select area component and set **NumbersOnly** checkbox to checked.

What just happened?

Okay, pretty cool the way our error trap works. If the user's entry in our radius textbox is a number, **then-do** is effected and the area gets calculated just like before. But, if a letter is included by mistake (like fumbled-fingerness—an invented term, but it fits) the **else-do** kicks in and our app gracefully recovers by clearing the erroneous entry and asking for another entry.

Here's the complete set of blocks for *Pi Are Square*:


def area as number 123
to area_of_circle arg name pi arg name radius arg
do set global area to value pi × value radius × value radius
return global area
when Calculate.Click
do ifelse test call is a number? thing radius.Text
then-do set area.Text to call area_of_circle pi number 3.1415926535 radius radius.Text
else-do set radius.Text to text set area.Text to text
when Clear.Click
do set radius.Text to text set area.Text to text

Okay, this was a lengthy example, but we learned several important concepts:

- Creating a professional-looking design with horizontal spacing
- How **procedureWithResults** works
- Building a function in App Inventor
- How to pass arguments to that function and receive and use the result
- Error-trapping to avoid app-killing mistakes

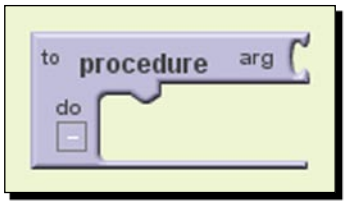
All of the previous techniques will be useful to you many times over in almost every app you create in App Inventor—especially the design part.

Now, we'll speed up a bit in looking at the rest of the blocks found in the drawers on the **Built-In** tab of the **Blocks Editor**. But, we are building the foundation you need (and, I certainly hope, want) in order to create great apps.

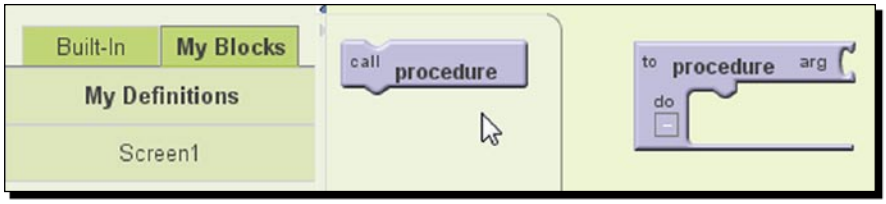
To whet your appetite, Chapters 5, 6, 7, 8, and 9 are all apps, all the time, and we'll be having some glorious fun in using all the basics in the first four chapters.

Procedure

A **procedure** is just like **procedureWithResult** with the exception that it does not return any result. Its sole purpose is to perform some function; for example, change the image of a component, or make them visible or not visible. Get a **procedure** block from **Built-In**/**Definitions**, and drag it out into the main area of the Blocks Editor, as shown in the next illustration:



Now, click on the **My Blocks** tab, and open (by left-clicking on its name) the **My Definitions** drawer. You will find that the mere process of pulling out the **procedure** block automatically created a call procedure block in the **My Definitions** drawer as shown in the following screenshot:



The previous is good. You can rename the **procedure** framework on the main area of the screen (left-click on the name, edit it to whatever you like, except use underscores (_) for spaces.

The **My Blocks** tab (to emphasize this point once more) is very important to you. This is where the magic of App Inventor really occurs as it automatically—based on the components you pick in **Design** or by bringing out several of the built-in blocks such as **procedure**—creates the blocks you'll need to make your app work.

Basically, a **procedure** framework block allows you to collect a group of blocks together, making (as we learned earlier) a subroutine that can be called by its name and used over and over.

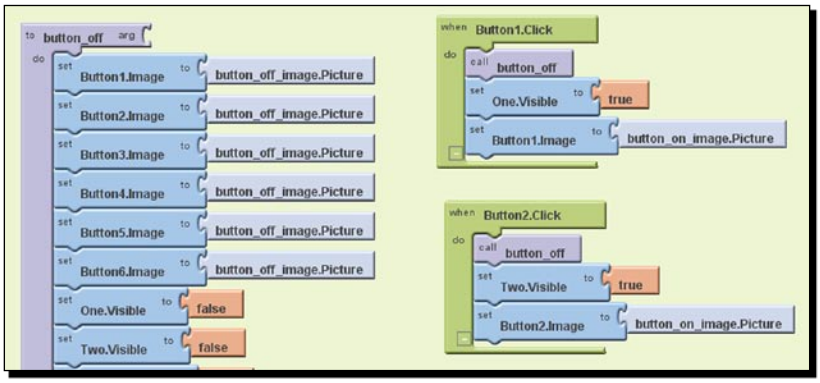
Unlike our previous example (*Pi Are Square*), in which the `procedureWithResult` was sent arguments and returned a result, **procedure** is most often used without an argument, despite the socket being there. This lets you pass arguments and do stuff, but does not return any result.

The following is an example of using a subroutine to make things easier. I wrote an app template that has six virtual screens and buttons that change, indicating the active screen.

Download from Wow! eBook <www.wowebook.com>

My logic for this app is that when any of the six screen buttons are pressed, all screens get set to off, and all buttons display their 'off' image. Then, the active screen is made visible, and the active button changes image to reflect that state.

I use a **procedure** framework renamed to **button_off** to do all that turning off for all buttons, as shown in the following screenshot. Then, just call it using the **call button_off** block found in the **My Definitions** drawer. In short, built the blocks once, used them six times.



We'll be using **procedure** a lot in the apps we build in this book, so it will become a trusty old friend.

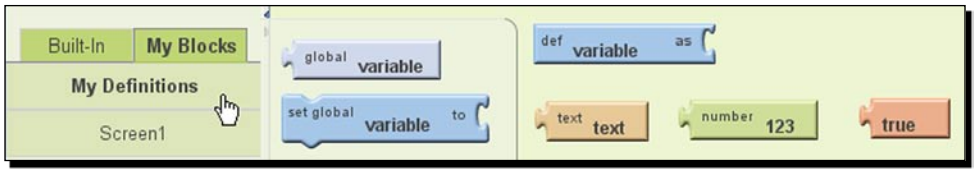
Variable

We've already used **global variable** several times. A variable in any programming language holds values—often a number or a string of letters, spaces, or words. Variables in many languages are declared; that is, to use them, the program must know what sort of data they have in them. Global variables may be used anywhere in the program.

All the previous applies to **global variable** blocks in App Inventor. In the following illustration, I've slid out a global variable for you from the **Built-In/Definitions** drawer. This creates two blocks in the **My Definition** drawer. Use the top one to get information from and the bottom (the one that includes an open plug) to set the variable's value.

To use a variable in an app, you name it (left-click on variable, change it to the desired name) and declare its type by plugging in the appropriate block. I've pulled the three possible ones out—text, number, and logic (`true` or `false`) as shown in the next screenshot.

Get these blocks from the **Built-In Text**, **Math**, and **Logic** drawers respectively. You do not have to change the default value when declaring.

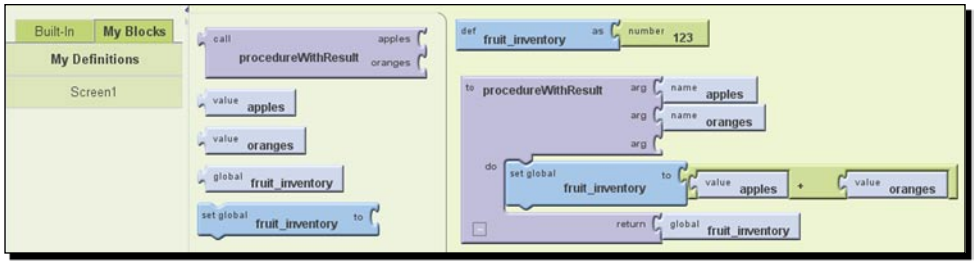


Name

As we learned in our *pi*-baking adventure previously, the `name` variable is a localized value used in an argument (a value passed to a specific routine). It differs from a global variable in which its value is only available in the procedure.

Here's an example showing a fruit inventory app. Any time we wish to update the global variable containing the total number of fruit, we pass the current numbers of apples and oranges as arguments to the procedure. The local addition of these two arguments changes the global value of **fruit_inventory** (making the new total available anywhere else it needs to be used in the app).

Note that in the **My Definitions** drawer, each name variable you pulled out and named has a corresponding local value block, which are used in this example.

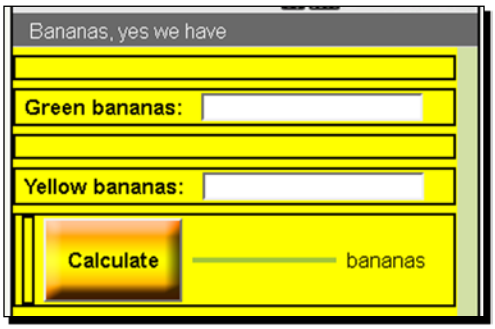


As the old song goes, "Yes! We have no bananas today".

But, what if we did have bananas? What if we had some yellow and green bananas and wanted to keep track of the total number of all bananas? How would we input and use these values?

Here's just a sketch (rather than step-by-step) of the procedure to create a banana-counting app.

In **Design**—using `HorizontalArrangement` for spacing and to put elements side by side—we need two textboxes for input, a calculate button, and a label for the total. Plus, of course, the three banana labels just to explain things (that is, label them). It should look something like the following:



In the **Blocks Editor**, we create the logic (and thus demonstrating how you use local values).

The **banana_inventory** global variable tracks our total number of bananas as shown in the next screenshot.

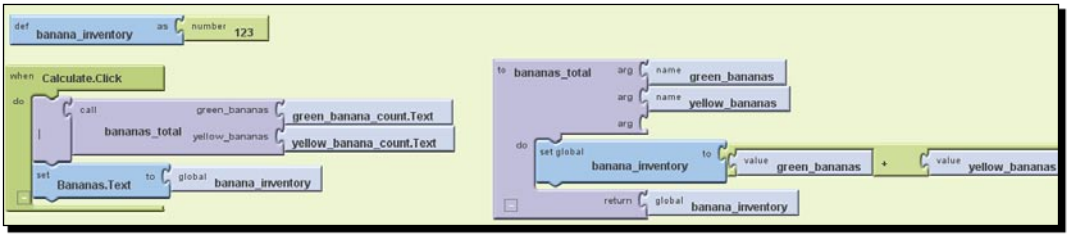
I've created a `procedureWithResult` and named it **bananas_total**. It uses the two name variables **green_bananas** and **yellow_bananas**.

In **Calculate.Click** (the button framework that controls what happens when the Calculate button is pressed), I put the call to the **bananas_total** function (use a **Dummy** block from **Built-In**/**Definitions** to make it fit). Wait for it, here comes the magic. The two textbox variables containing the input—**green_banana_count.Text** and **yellow_banana_count.Text** (get them out of the automatically-created drawers for each textbox in the **My Blocks** column) are snapped in as the argument sockets for our **name** variables **green_bananas** and **yellow_bananas**. As you can see in **Calculate.Click** shown in the next screenshot, those names are already there.

What we all have to get used to is App Inventor doing most of the work for us. All we have to do is drag blocks into place. Wonderful once understood and accepted.

So now the localized arguments go to the function, are added together, and the global variable **banana_inventory** is updated.

Here are the complete blocks for the app:



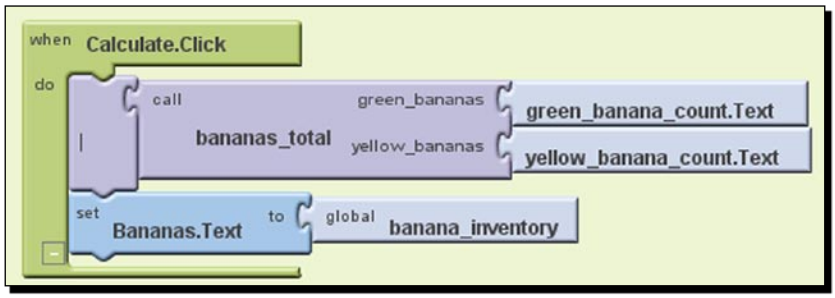
And what it looks like on a phone:

And that's how you get real-world input into and use `name` variables.

Dummy

This block does nothing and does not generate anything in **My Blocks**. It's simply an adapter for when you can't make a block go into a socket otherwise, and looks like the following:

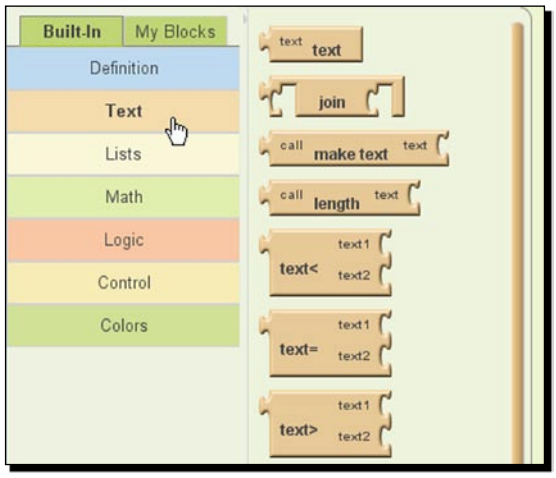
The previous banana-counting app is an excellent example of how this block is used. My call to the **bananas_total** function would not click in, so I used a dummy block. Works great.



Now, let's move on to the next drawer on the Built-In tab of the **Blocks Editor**—**Text**.

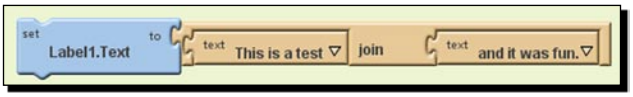
Text Blocks

In the **Text** drawer, we find blocks related to manipulating strings—`strings` being letters and spaces—in other words, *words*. There are currently 19 blocks in the **Text** drawer (see the top few in the following screenshot). We buzz through these for you with examples as warranted.



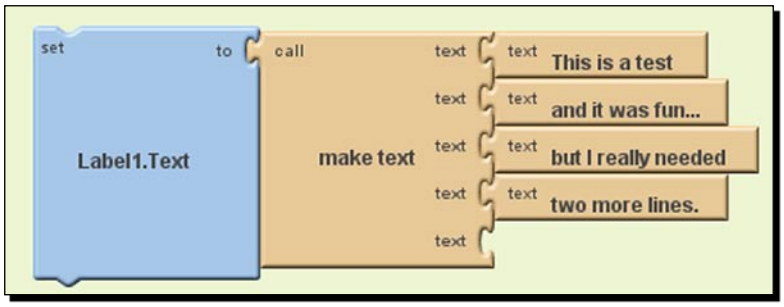
text: This block, the top one in the drawer, holds strings, and we can cram thousands of words in just one of these, so it has practically no limit so far as using it in an app is concerned. To put text in it, right-click on the lower **text** below (the word `text` is the default value), and you can change it to whatever you like.

join: Joins two strings. Use it as shown in the next screenshot, which produces a label reading: `This is a test and it was fun.`, and put a space at the end of **test** so that it and **and** will not run together.

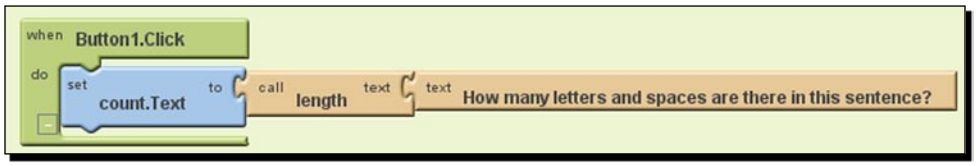


By the way, the small triangles on the previous **text** blocks will often appear on blocks—they're simply shortcuts so that you can change the type to a related type. For example, in the case of **text**, it can change to a **number** block and vice-versa.

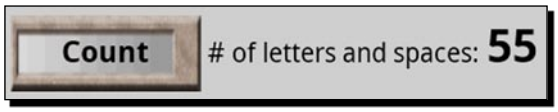
make text: Is more powerful, letting us add additional lines as the following screenshot shows. It's also very useful in building e-mail headers and the like as we'll see later on. Like we found earlier in adding arguments to a procedure call, every time you add a line, a new socket appears—but you will eventually hit a limit, so try to avoid overly complex blocks (that is, use two or more if needed and combine results).



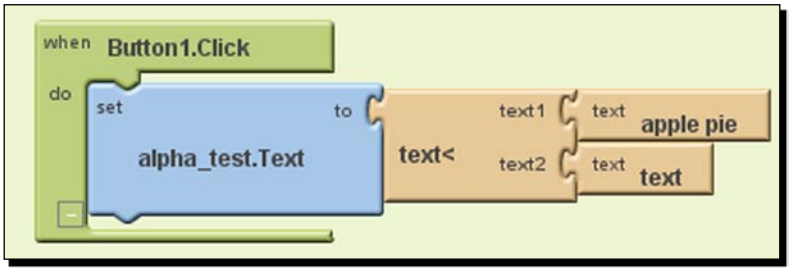
length: Returns the length of a string as shown:



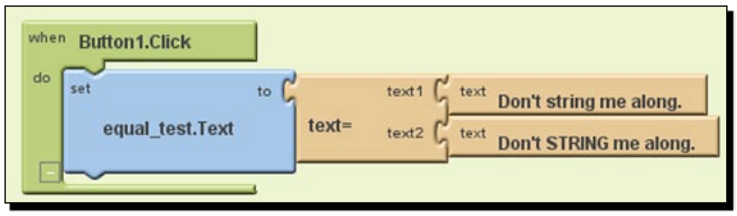
This might look something like the one shown below on your phone, depending on the cosmetics you added in **Design**.



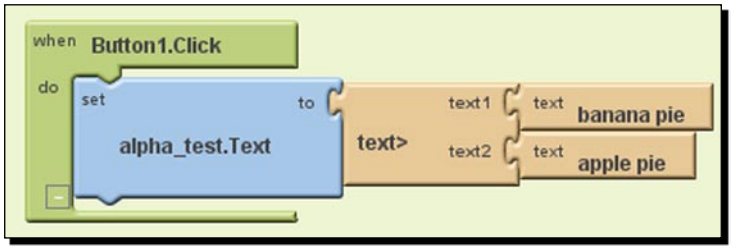
text<: Tests the alphabetical order of two strings. In the following example, the value of **alpha_test.Text** would be `true`. However, uppercase letters come in front of lowercase, so if we change **text** to **Text** below, the result is false; that is, the two strings are not in alphabetical order.



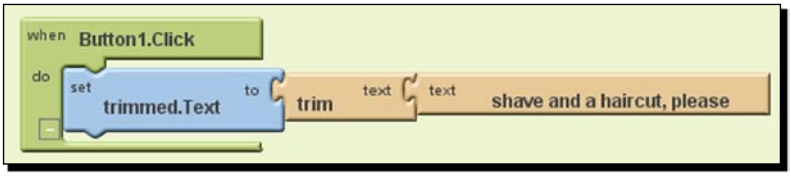
text=: Compares two strings to see if they are the same. In the following screenshot, the strings are different so `false` is returned (that is, **set** as the value of **equal_test.Text**).



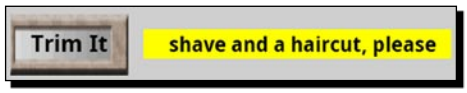
test>: Tests for reverse alphabetical order, so here we get `true` because `b` comes before `a` in reverse alphabetical order.



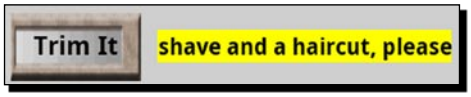
trim: Snips off leading and/or trailing spaces. This is how you can use it:



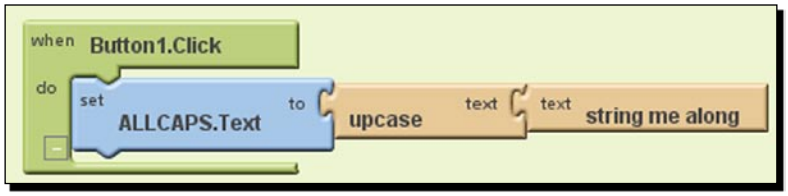
On your phone, it could have looked like this before:



And after as so (with spaces "shaved" off):

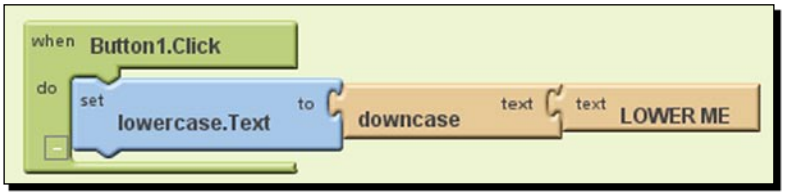


upcase: Changes every letter in the string to uppercase:

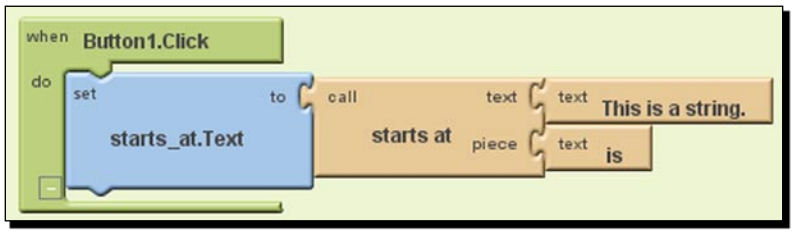


Here's how the result could look on your phone:

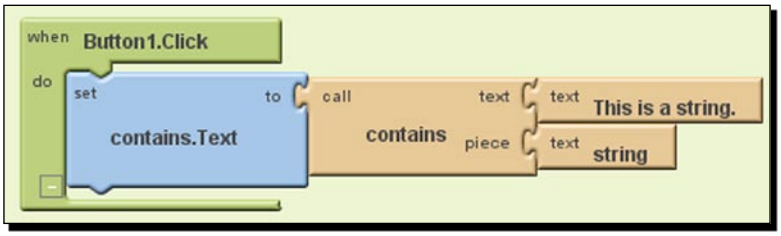
downcase: Changes every letter in the string to lowercase:



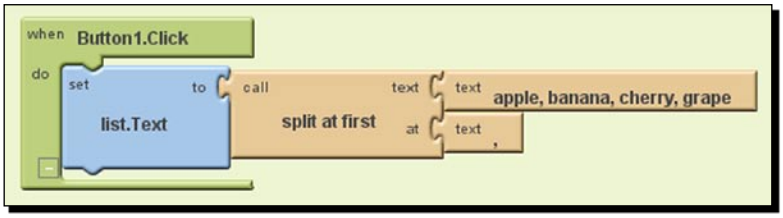
starts at: Takes two arguments, a string and a piece of that string. In the following example, the piece is the word **is**. So, the value of **starts_at.Text** is set to `5`, because it starts at position five in the string (don't forget to count the space). If the second argument does not appear in the first, the value of `0` is returned.



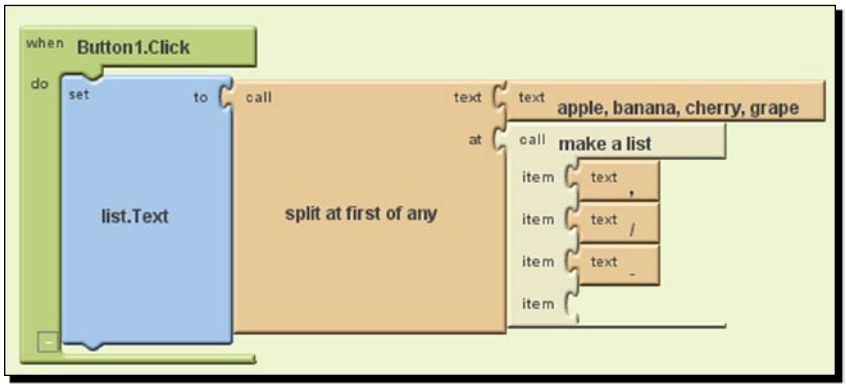
contains: Also takes two arguments, but check to see if the second string (the **piece** argument) is present in the first. If the search string (**piece**) is there, `true` is returned or else `false`.



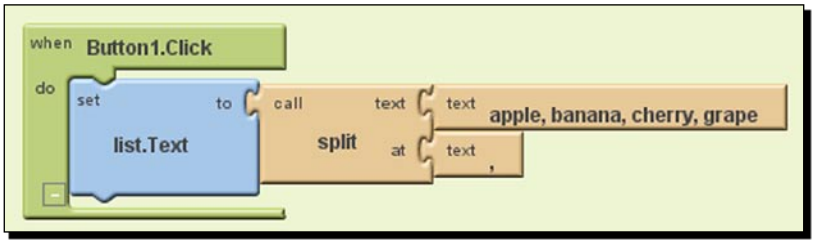
split at first: Splits the string in the **text** argument into a two-element *list* (we look at lists —which are both powerful and useful—in the next section). The split is at the first occurrence of the character or string in the **at** argument (which is called a *delimiter* in lists, being something that separates strings into a list—commas are often used). In the following example, the string splits at the first comma (after `apple`). The comma is dropped, and the two elements of the list are `apple` and `banana, cherry, grape`.



split at first of any: Works as the **split at first** except the **at** delimiter is a list itself—see the following group of example blocks where we have a list of three delimiters (comma, slash, and hyphen). Since the last two in the **at** string are not present, the string still splits at the first comma (after `apple`). The comma is dropped, and the two elements of the list are again `apple` and `banana, cherry, grape`. If none of the delimiters appear, then the original string is returned unchanged (which means you do have a list yet).



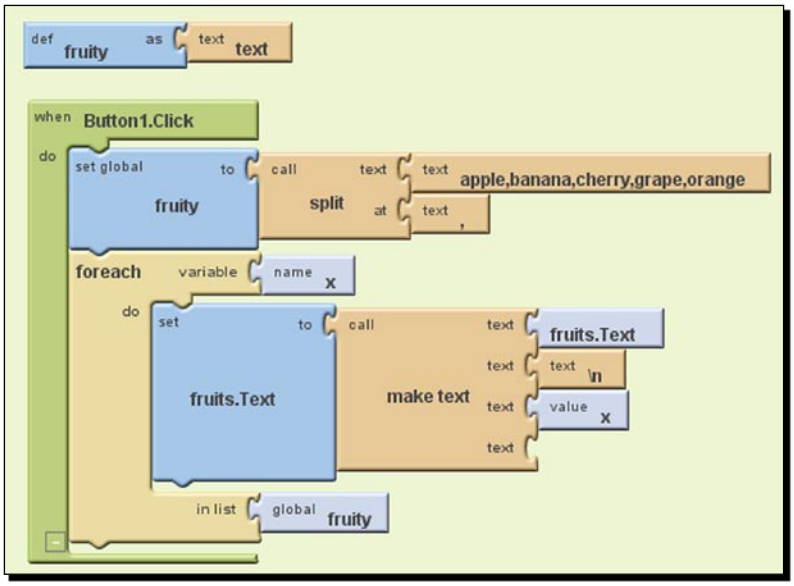
split: All delimiters in the **at** string (a comma in this case) are acted on. So, using the blocks shown in the following screenshot, a four-element list is returned (and the commas are dropped).



You might be asking how do I display a list on my phone once I have it? Well...

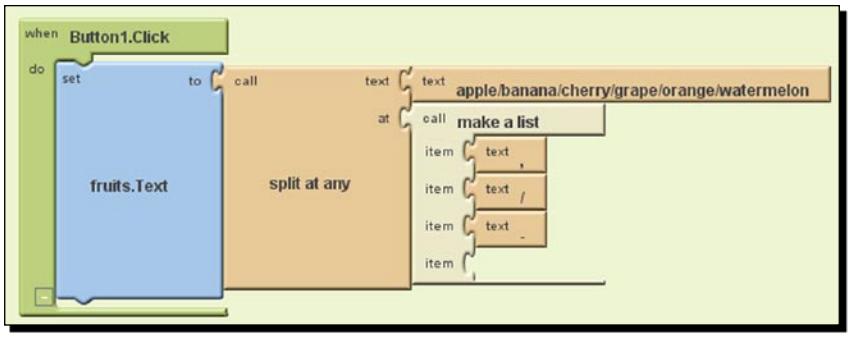
Have a go hero – displaying a list

1. In **Design**, you need only a Button and a Label component. Fancy it up with `HorizontalArrangement`, colors, and so forth as you like (we learn by playing!).
2. In **Blocks Editor**, duplicate these blocks (hint: get the **foreach** framework block from the **Built-In/Control** drawer, and use `\n` as a line break).

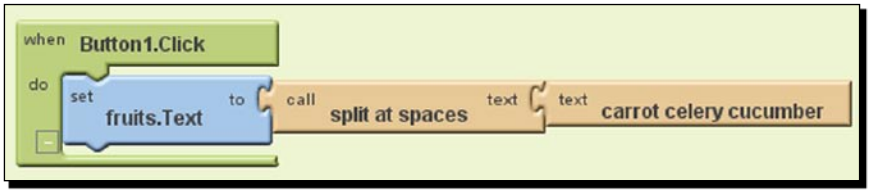


You'll get something like this on your phone:

split at any: As with **split at first of any**, a list is used for the **at** argument. In the following example blocks, the fruits are separated by slashes as a delimiter (which is in the list), so a complete list of fruits will be built in **fruits.Text**.

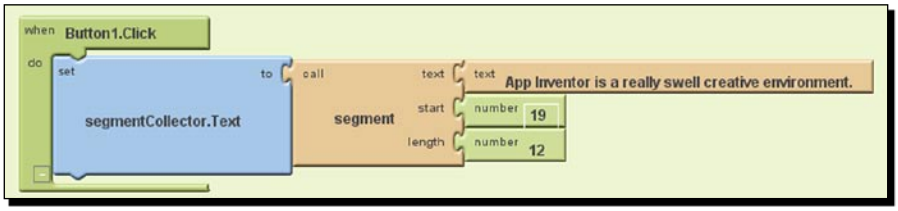


split at spaces: In the string of three vegetables as shown in the following screenshot, this block causes them to be split at all spaces, and the spaces get dropped. So, we wind up with a three-element list of vegetables with **carrot**, **celery**, and **cucumber** as its elements.



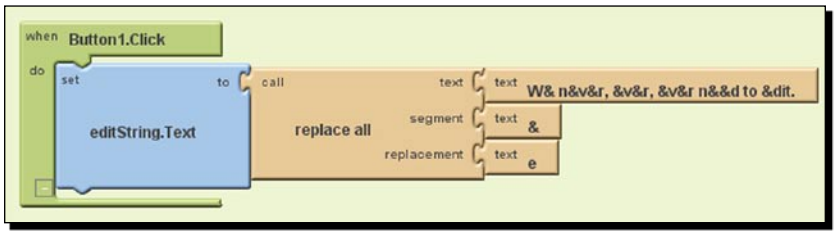
segment: Returns a segment or portion of a string. The three arguments are the string, the index number to start, and how many spaces to pull out. In our example, we start at position 19 and take out the next 12 positions.

You can count them if you like, but the result is `really swell`.



replace all: The final block in the **Text** drawer replaces characters or strings. In the following example, we have a string in which a much-needed vowel has been replaced by ampersands. This is not good. Let's fix it.

The three arguments required for this block are the string, the text to be replaced, and the replacement text.



And here's the result on my phone:

Yeah, right.

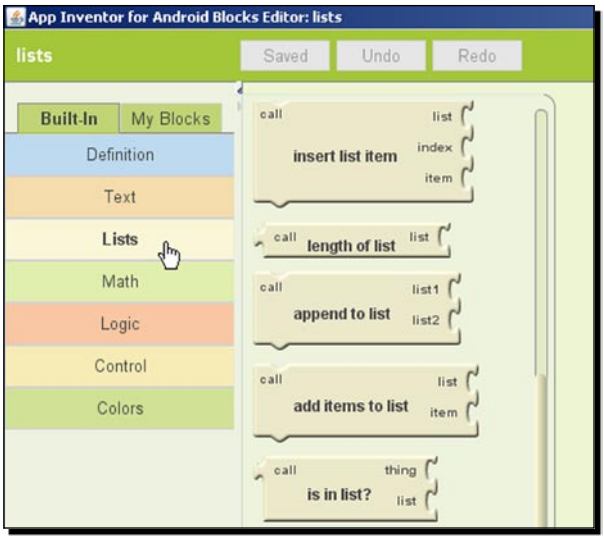
Now, let's look at lists in deeper detail as we explore the **Built-In/Lists** drawer.

List blocks

We'll often need to use *lists* of things in creating apps.

Lists are collections of simple data. `Apple, cherry, grape, orange` is a list of fruits. The e-mail addresses of your friends are a list. Often-called phone numbers could be another list you need an app to retrieve.

We've already met the list concept previously in playing with some of the blocks from the **Text** drawer devoted to assisting you in using lists in App Inventor. But, the **List** drawer carries this further, giving us 18 blocks for manipulating lists (currently 18; the developers keep adding goodies). The following screenshot shows the **Built-In/List** drawer open:



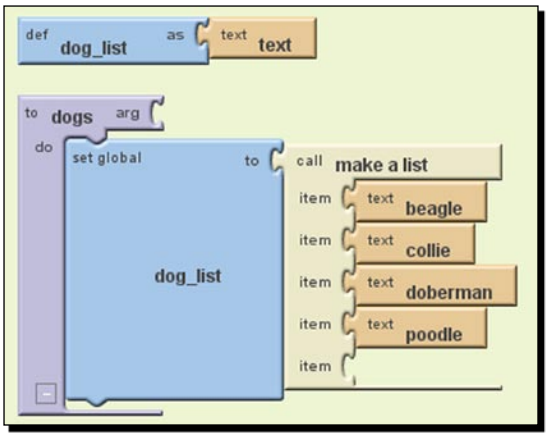
Next are some examples of using these blocks.

First, however, we need to make a list to work with—this one for types of dogs. Define a global variable—usually declaring it as a text variable for holding lists. I'm calling mine **dog_list**. You can either attach a **make a list** block to it now or when the list is actually added.

make a list: Which brings us to using the first block in the Lists drawer. I like putting my lists in a procedure framework because it makes them permanent from session to session, so I pulled one out and named it **dogs**. From the **Built-In**/**Definitions** drawer, pull out **def variable as**. Change the variable to **dog_list**. Now, open the **My Definitions**. You will find that **global dog_list** and **set global dog_list to** have been added. Snap the **dog_list** variable just defined into the **dogs** procedure framework. Then, from **Lists**, drag out **make a list** and click it into the variable.

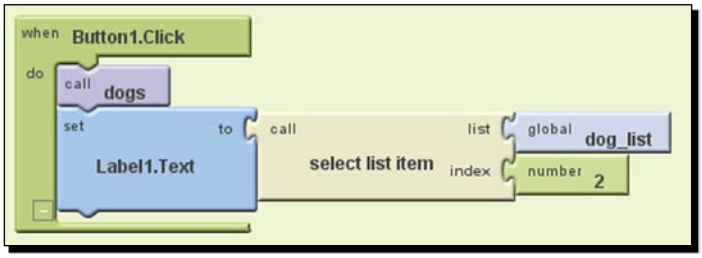
Later, I will show you how to make your lists more dynamic by having them save changes made during each session.

Pull out Text blocks (top one in the **Built-In**/**Text** drawer), and populate the list with dog types.



Lists can be created manually as we just did or automatically, as we did by splitting out elements of the list from text strings with the **Text** block tools earlier.

select list item: Displays a single item from the list. We use the number `2` in the following example to show the second item in the list. But, first, we call the procedure **dogs**, which loads the list into our global variable, **dog_list**.

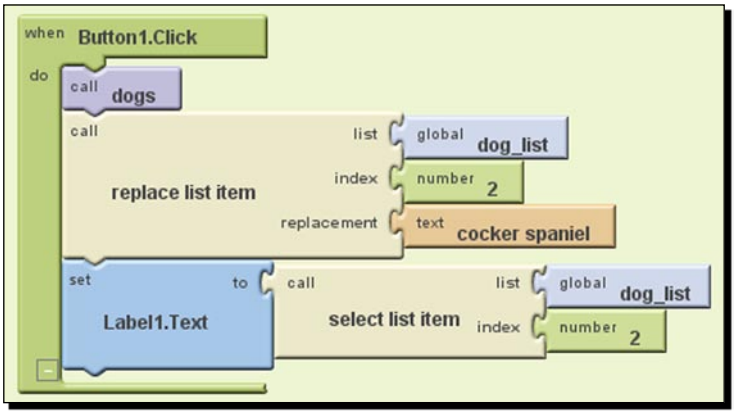


In my dogs list, the second item is `collie`, and it looks like this on the phone.

Download from Wow! eBook <www.wowebook.com>

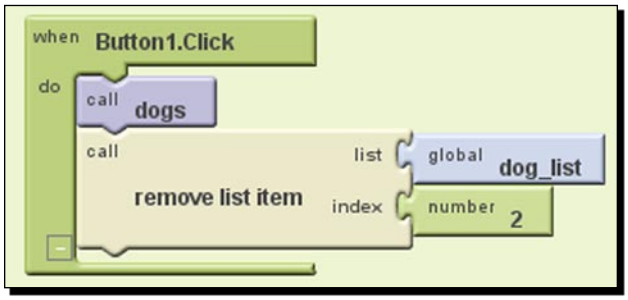
Remember those free web design buttons on the Internet? They often have matching dividers with them. I used them to spiff up the app design previously—inserting them as **Image** components and using `HorzontalArrangement` for spacing.

replace list item: Replaces a specific item in a list. In the following screenshot, we replace #2, `collie`, with `cocker spaniel`.

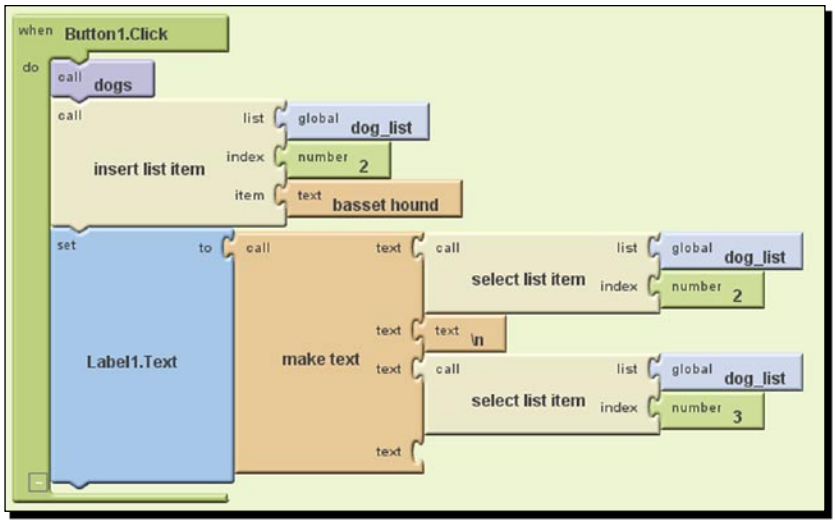


This shows when I press the **Get Dog** button on my phone.

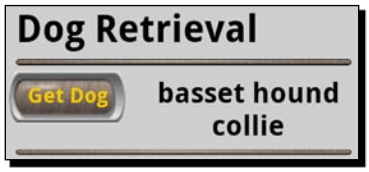
remove list item: Removes the specified number item from the list. In the example, we remove `collie` completely.



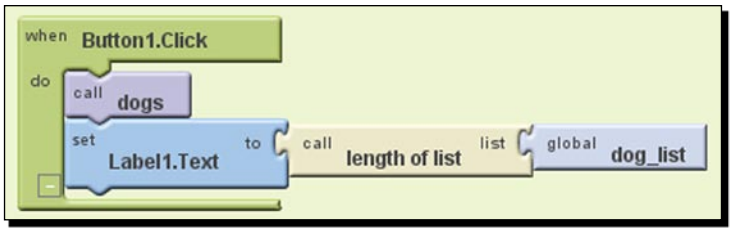
insert list item: Inserts a new item in the list at the position requested in the **index** argument. In our following example, we insert `basset hound` at `2` and ask for retrieval of list items `2` and `3` (we include `\n` when displaying elements of a list to insert a hard line break).



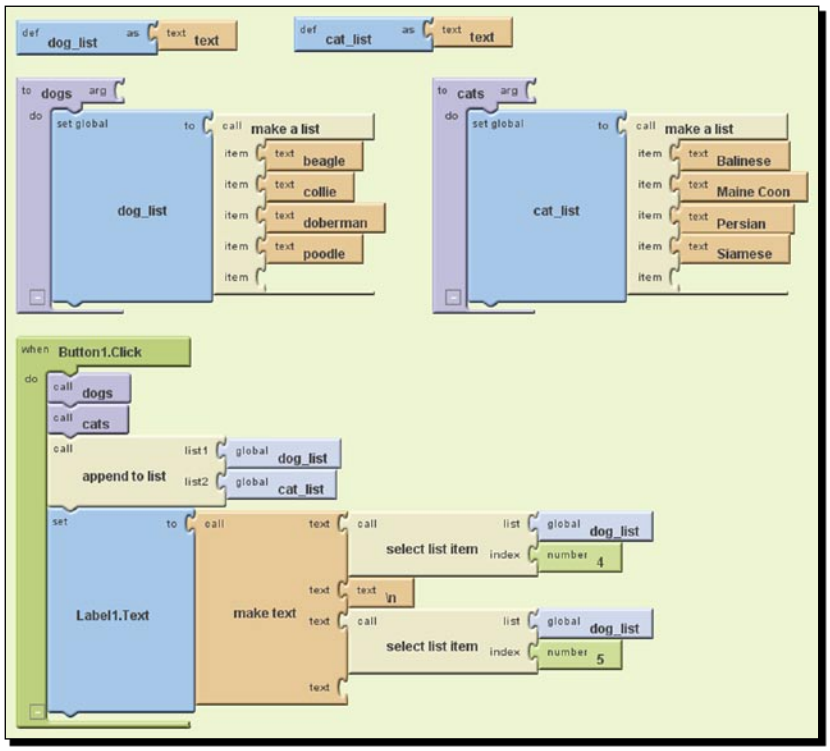
And it looks the following on my phone:



length of list: Counts the elements in the list and returns the number. If the list is empty, it will return `0`.



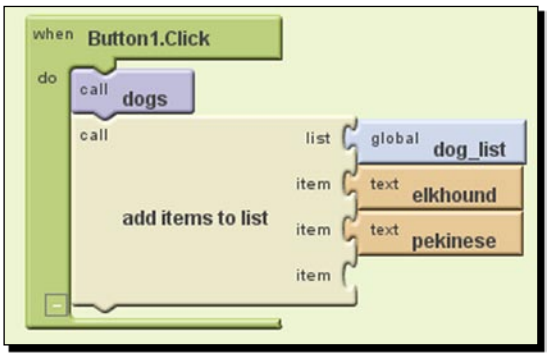
append to list: Appends a second list to the first, without changing the second list. In our continuing example below, I've added a **cat_list** to it, and used **append to list** to join the two lists. When the button is clicked, the last position of the original dog list, `4`, will display as well as the next position, being the first from the cat list—now item `5` in the dog list.


def dog_list as text text
def cat_list as text text
to dogs arg
do set global dog_list to call make a list
item text beagle
item text collie
item text doberman
item text poodle
item
to cats arg
do set global cat_list to call make a list
item text Balinese
item text Maine Coon
item text Persian
item text Siamese
item
when Button1.Click
do call dogs
call cats
call append to list list1 global dog_list list2 global cat_list
set Label1.Text to call make text
text call select list item list global dog_list index number 4
text text \n
text call select list item list global dog_list index number 5
text

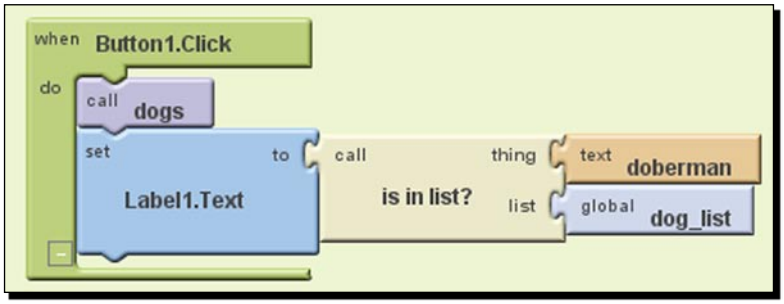
This append is not permanent since we have both dog and cat lists in procedures, so you still have the original of both in your app.

Here's what it looks like on my phone. Note that I updated my labels to reflect the addition of felines to the app. Also good to be thinking of the end, user and keep your design up-to-date.

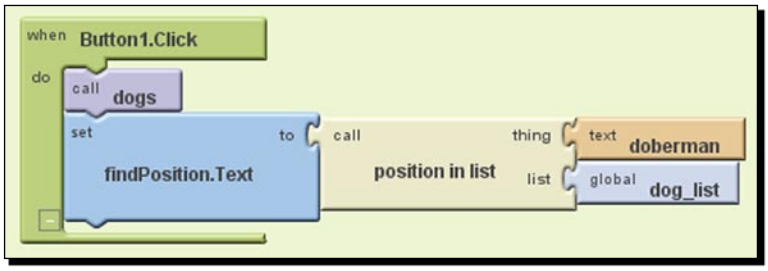
add items to list: Adds one or more items to a list as shown in this example:



is in list?: Returns `true` if the search term in the argument is present in the list. Here, doberman is there, so the result is `true`, otherwise would have returned `false`.

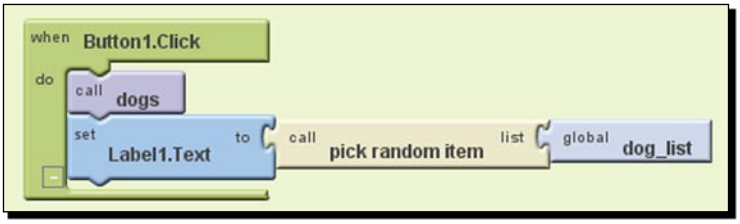


position in list: Lets us search for an item to find its position in the list. Using it the way I did in the next illustration lets us find and display an item in a list when we do not know its actual number—which will be almost always for a list having lots of changes such as insertions, deletions, and so forth during a session.

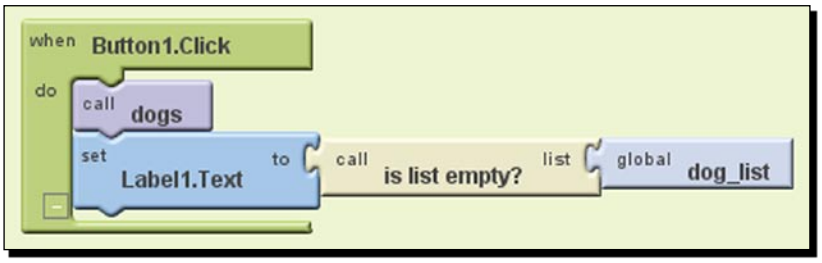


So my phone shows this when I press the **Get Position** button:

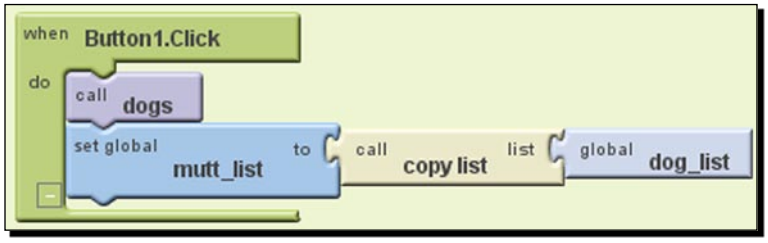
pick random item: Displays a random item from the list and can be used like the following:



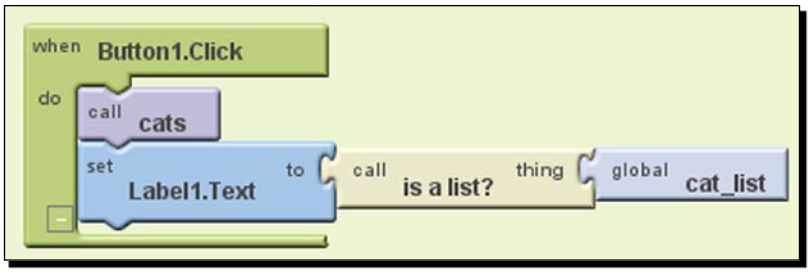
is list empty?: If no items are in the list, `true` is returned. In the following example, we get `false` because there are dogs in our dog list.



copy list: Copies a list into another variable, creating a list structure and used as the following:



is a list?: Checks to see if the variable contains a list and can be used as following. Having created it just recently, we know **cat_list** is a valid list—this block confirms that by returning `true`.



CSV list blocks

With the most recent update (April 19, 2011) to App Inventor at the time of writing, the **Lists** drawer got four new blocks. These are for *CSV* (comma-separated values) operations. CSV is a *lingua-franca* (common language or format) between many types of computer software such as spreadsheets (Microsoft Excel, for example) and databases. It allows export of data from one program to another.

CSV data is in the form of rows—*item*, *item*, *item* with a line break ending the row. *Tables* are several rows all in the same format.

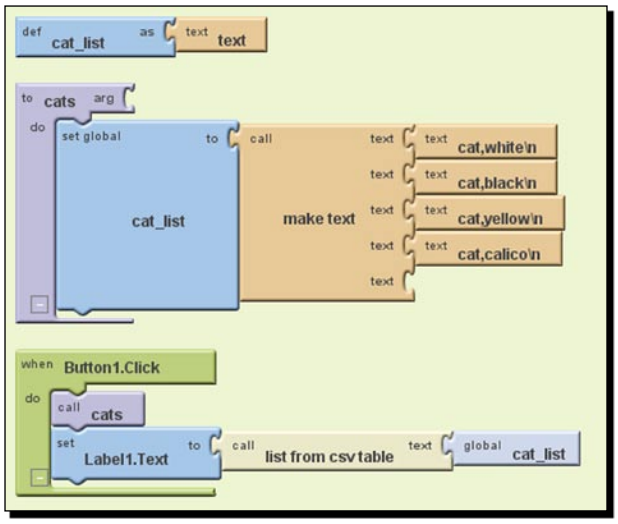
In the **Built-In/Lists** drawer, we now have four blocks to use with CSV data:

- list **to csv row**: converts the complete list to one row of CSV data.
- list **to csv table**: converts a list to a table by inserting line breaks between each item in the list. Each item is itself a list representing a row of the table.
- list **from csv row**: converts a row of CSV data to a list, with each comma-separated item becoming an item in the list.
- list **from csv table**: converts a CSV table (file) to a number of lists, with each list containing the items from a row.

We will be using these four blocks a lot later in the book, especially in apps using Google's online database service called *Fusion Tables*.

Next is a quick example of a list from a table. The result is a list with *sublists* for each row.

In the following **cats** procedure, I've changed the list to a regular text variable with rows (comma-separated items) ended by a line break (`\n` in App Inventor).



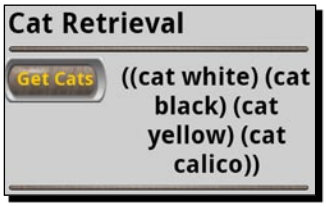
So, what we now have in **cats_list** is equivalent to a CSV file—comma-separated data in rows and line breaks ending rows.

In the **Button1.Click** framework we put what happens when the button on the phone screen is clicked (**Get Cats**, next).

Referring back to the previous screenshot, we:

1. Call the **cats** procedure to load the variable **cats_list**.
2. Use the **list from csv table** block to convert the data into a lists-within-a-list format usable in App Inventor.

Pressing the button, we get the lists in a list result as shown in the following screenshot:



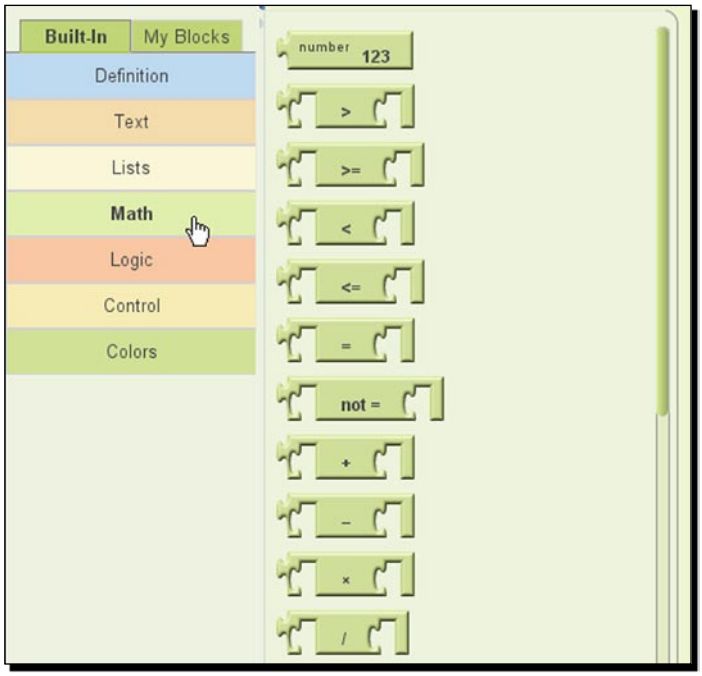
Later, I'll show you how to format this result into neat tables as the mileage log app I'm working on and mentioned earlier. It pulls CSV data from Google Fusion Tables and converts it using the technique we just looked at.

Vehicle	Start	Stop	Type	Date	Miles
Kia	30000	35000	Personal	4/26/11	500
Target	68000	69000	Notary	4/30/11	1000
Trumpet	42000	43000	Notary	4/29/11	1000

Now, let's do some math.

Math blocks

The **Math** drawer of built-in blocks shown in the following screenshot is the longest, currently having 39 blocks. Let's take them by blocks of similar operations just to speed things up a bit.

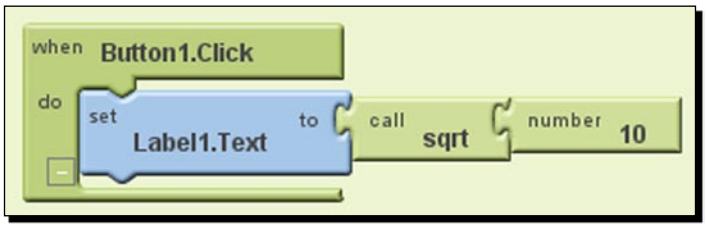


The top block in the **Math** drawer is one we've employed often already in the example apps. It's one of the blocks you'll use the most, **number 123**. This block is simply a number holder. The default value is `123`, but just left-click on that figure and you can change it to whatever you like.

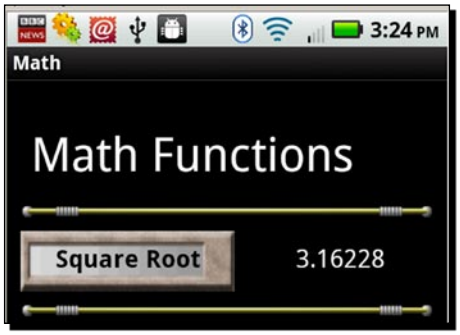
Math frameworks: The next 10 blocks are frameworks. In the previous screenshot, the first six framework blocks (from top to bottom) return (`true` or `false`) the result from comparing two variables and/or numbers that is greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal to.

The final four blocks do simple math operations on the two variables and/or numbers plugged into them—add, subtract, multiply, or divide.

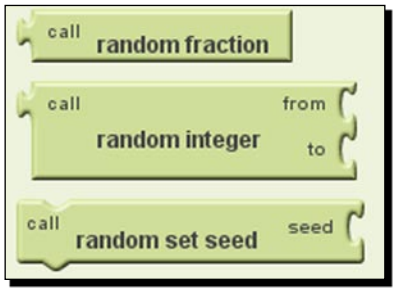
sqrt: This block calculates the square root of a number and is used like this (square root of `10`):



The answer on the phone might look like this:



Randomizing: Three blocks—**random fraction**, **random integer**, and **random set seed** (shown in the next screenshot) are useful in obtaining random numbers, such as in writing game apps (just click them into a numeric variable to use). The first returns a number between `0` and `1`. The second allows you to specify a range using the **from** and **to** arguments, such as between 1 and 100. And the last sets a random seed—useful in making sure numbers really are random.

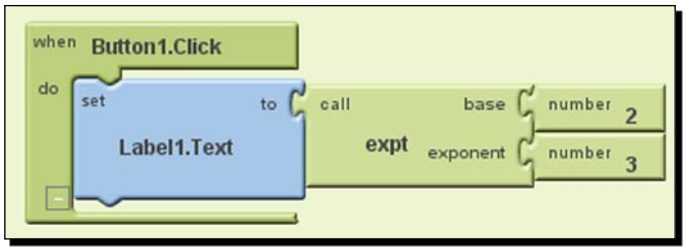


Looking at the next few blocks, **negate** makes the number plugged into it a negative value. Several values can be plugged into **min**, which returns the least or minimum value. And max (or maximum) reports the highest value of the numbers or numeric variables clicked into its ever-expanding sockets.

Division: Next are three blocks relating to the division process—**quotient**, **remainder**, and **modulo**. The quotient is the number divided into a number, such as in `20` divided by `3`, `6` is the quotient. The remainder would be `2`—that's what's left over. The modulo is the same as the remainder when both the number divided and the quotient are positive, but gets a bit more complicated when one of these numbers is negative.

Rounding: The next four blocks—**abs**, **round**, **floor**, and **ceiling**—have to do with nearest whole values of numbers. The absolute value of a number is returned by **abs** (makes negative numbers positive). Rounding to the nearest whole number is done with **round**. Finally, **floor** truncates to the nearest integer (whole number) that is less than or equal and **ceiling** to the nearest integer greater than or equal.

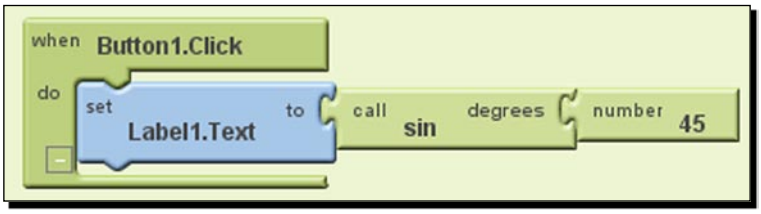
expt: The name of this block stands for exponent. Using an exponent means raising a number to power of itself or multiply itself by itself that many times. In our example, we cube the number `2` or `2` x `2` x `2`. The answer returned is `8`.



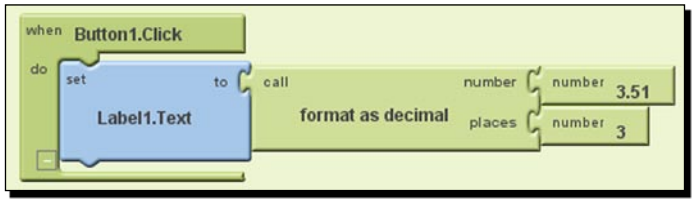
Log and e: the **exp** block returns the constant **e** (2.71828...) raised to the power of the given number, and **log** calculates the natural logarithm of the given number.

Trigonometric functions: The next nine blocks are all trigonometric functions—**sin**, **cos**, **tan** (sine, cosine, and tangent); **asin**, **acos**, **atan**, **atan2** (arcsine, arccosine, arctangent, and calculating arctangent using rectangular coordinates); **convert radians to degrees**; and **convert degrees to radians**.

Here's how these trig functions might be used. The following example finds the sin of 45 degrees, which is `.70711`.



format as decimal: Lets us set the number of decimals to display. In our example, that's set to `3` places, so the number `3.51` displays as `3.510`.



is a number?: Finally, the last block in the **Math** drawer simply checks a variable and returns true if it's a number.

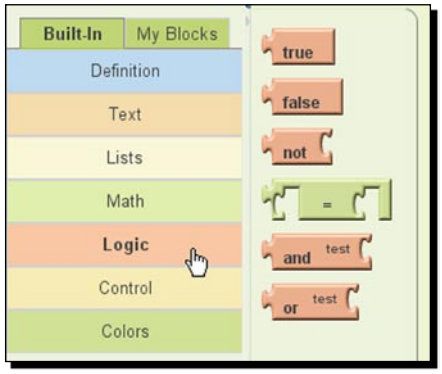
Now, we proceed logically to the **Logic** drawer. True?

Logic blocks

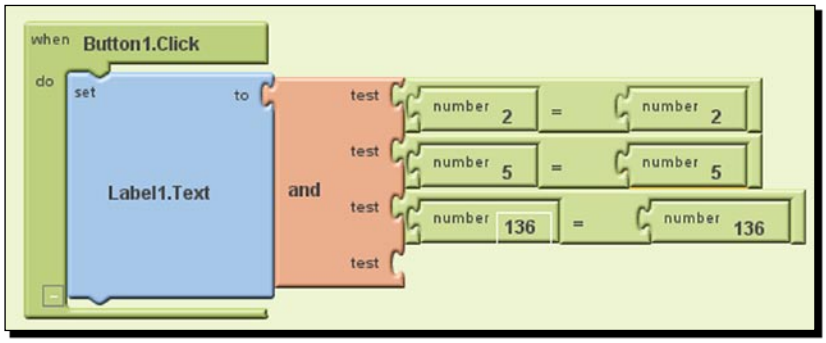
The use of logic is in testing various things to see if they are `true` or `false`. The first two blocks in the Logic drawer (shown in its entirety in the following screenshot) should be familiar by now—we've used **true** and **false** blocks several times in examples already.

The third, the **not** block, reverses logic—if something plugged in to the block is `true`, the **not** block reports it as `false`.

The **equal** framework works exactly the same and is interchangeable with the one in the **Math** drawer.



Using the **and test** block will return `true` only if all the things plugged into it are true. In our example here, all are, and we get `true`.



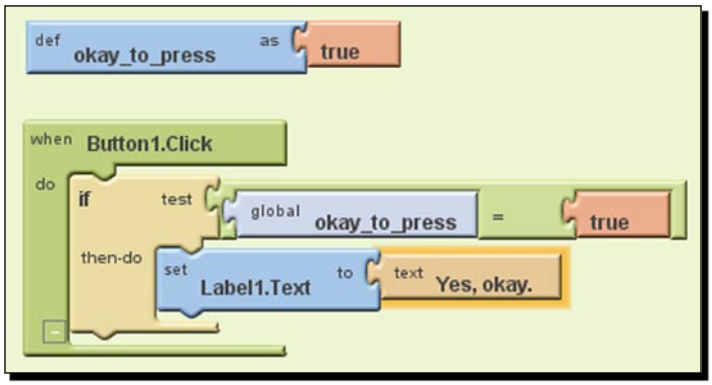
Finally, the **or test** block returns true if one or more of the items plugged into it are true.

Now, we take control of the **Control** drawer.

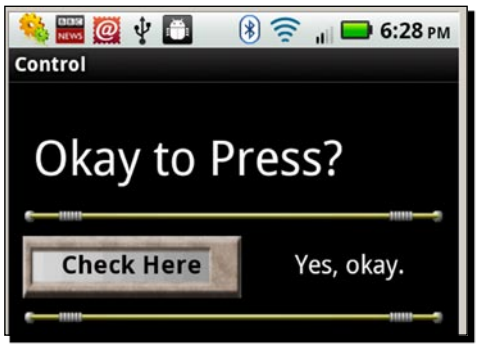
Control blocks

The **Built-In/Control** drawer contains blocks that, yes, control or act on events.

Take the first block **if test then do**—it's logic of response is simple but powerful. If a condition is true then do this; if it's not true, ignore. So, in the example here, the variable **okay_to_press** is `true`.



Pressing the following button tells us it was okay. If it wasn't, nothing would have happened.



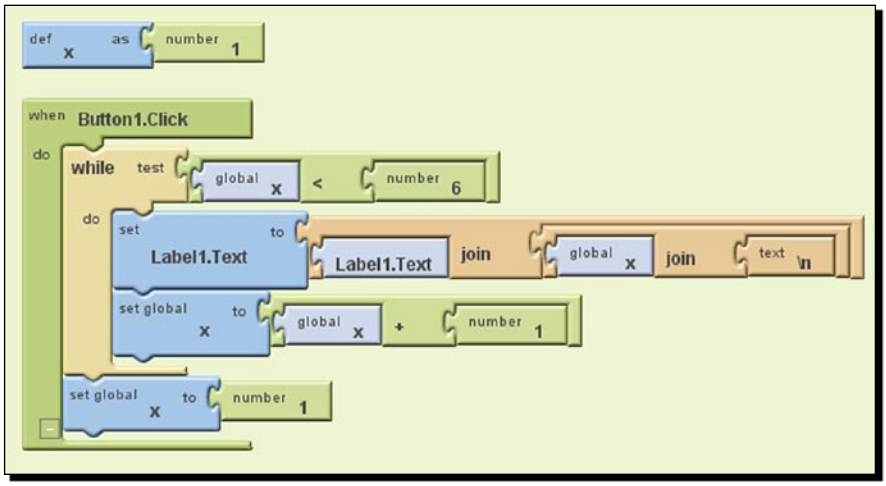
The next five blocks—**ifelse**, **choose**, **foreach**, **for range**, and **while**—offer more sophisticated ways of controlling events and responses in our apps. We'll be using those a lot and will go into detail on their operation several times.

But, let's knock off another basic programming concept, loops: A loop goes round and round until something happens and is one of the basic tools in any programming language. In App Inventor, the **while** block is quite nice for doing loops. Here's how: follow along on your computer.

Time for action – making a loop

Let's count to five.

1. In **Design**, you need a button and a label. Otherwise, fancy it as you like (remember, the more you format professionally, the more it becomes second nature).
2. From the days when I first learned to program using BASIC in the late 1970s on my first personal computer, I've liked using `x` as a counter in loops. So, we first declare a numeric global variable, `x`, in the **Blocks Editor**.
3. In the **Button1.Click** framework, place a **while** control block (this block repeats while a certain condition is true). Refer to the following illustration to see how the blocks should look.
4. For the **test** argument—since we're counting to `5`—make `x` less than `6` (that is, one more than `5`).


def x as number 1
when Button1.Click
do
while test global x < number 6
do
set Label1.Text to Label1.Text join global x join text \n
set global x to global x + number 1
set global x to number 1

5. In the **do** portion of the **while** control block, we need two lines of blocks. First, we double join to retain each level (`1-5`) and add the `x` count joined with a line break.
6. After the loop, we reset `x` to `1` so that the loop runs properly the next time the button is pressed. Be sure this is outside of the **while** framework otherwise you will have an infinite loop that will never end and will cause the program to Force Quit!

What just happened?

Pressing the button on our phone now causes a loop that displays each **x** on a line by itself for five times as shown in the following screenshot:

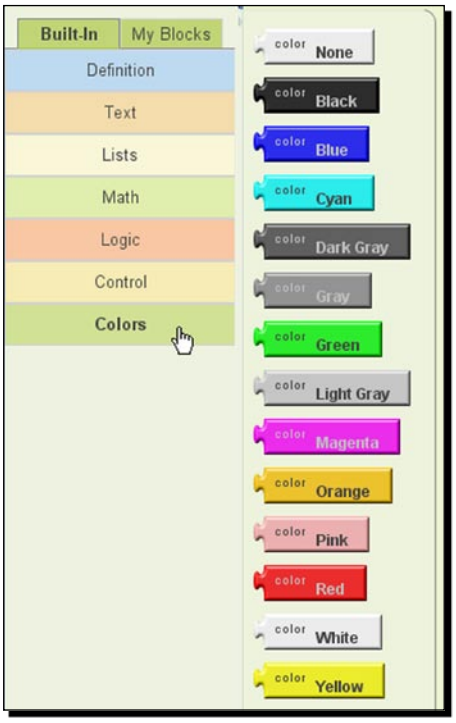
Download from Wow! eBook <www.wowebook.com>

Now we're programming. Oh... yeah... they don't like us calling it that. Er... now we're creating apps by moving blocks around.

But, the basics apply. Stick with me; I've got tons more tips like this. And speaking of writing colorfully.

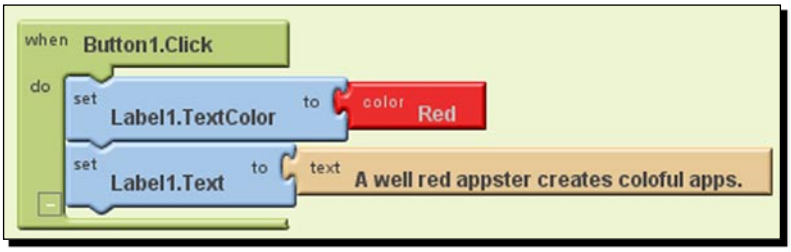
Colors blocks

The last **Built-In** drawer is **Colors**, as shown in the following screenshot:



Colors offers exactly the same colors as in the **Properties** column in **Design** for components such as the screen background, labels, and buttons that have a color property. But, with these blocks you can control the color of components, such as changing text from red to yellow, depending on whatever criteria pleases you.

In the following example, we change the text color to red, after pulling the **Label1.TextColor** block from the Label1 drawer on the **My Blocks** tab, which was created automatically when we first dragged-and-dropped the **Label** component onto the virtual phone screen in **Design**.



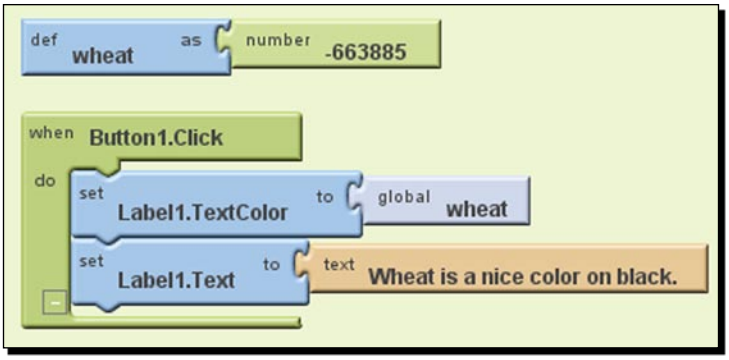
Pun, as ever, intended.

Nor are we limited to the 13 "standard" colors in the **Colors** drawer. You can define your own colors—thousands of them!

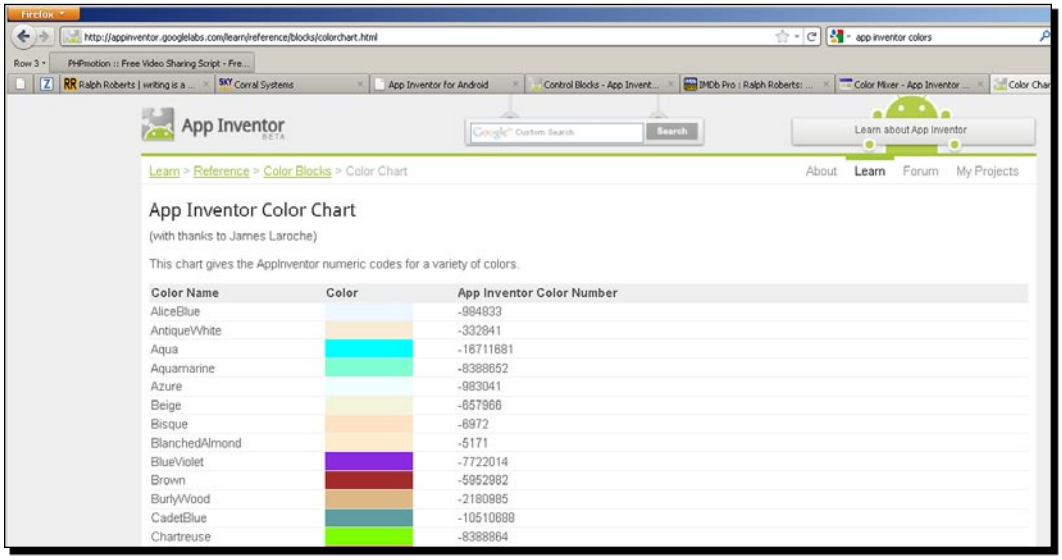
As in our following example, I simply declared a numeric global variable with the color number for the "wheat" named color. I got this number, and you can also, from the App Inventor site's Color Chart page (`http://appinventor.googlelabs.com/learn/reference/blocks/colorchart.html`).

Once the color variable is defined, it's automatically in the **My Definitions** drawer, and we simply click it into color block for the component we want to change the color of.

You could also just plug the color number in directly, but I think it is a bit more flexible to have it in a variable, thus allowing you to easily use it wherever desired.


def wheat as number -663885
when Button1.Click
do
set Label1.TextColor to global wheat
set Label1.Text to text Wheat is a nice color on black.

And here's how the Color Chart reference page looks:



That introduces us to colors. Now, a word about the **My Blocks** tab.

Individual component blocks

The **Built-In** tab of **Blocks Editor** always has pretty much the same blocks available. The "pretty much" caveat is there because each update (which you get automatically from the Google Labs developer heroes) adds new blocks from time to time.

Frequent the forums at `http://appinventor.googlelabs.com/forum` to keep up with all the new goodies.

The **My Blocks** tab is a totally different animal. The number of drawers on it depends on what components you've dropped in place in **Design**.

For every component you add in **Design**, a drawer for it is automatically created in **Blocks Editor**. We'll be looking at and using these drawers during all the many examples to come.

Now, a short quiz, and you may go out and play with blocks.

Pop quiz

1. Error trapping is:
    a. Setting out steel traps for huge bugs.
    b. Causes your phone to lock up.
    c. Blocks that prevent known errors.
    d. None of the above.
2. "Declaring" a variable is:
    a. Illegal in some countries.
    b. Requires practice speaking in front of crowds.
    c. Should be done firmly.
    d. Plugging a text, number, or logic block into a global variable.
3. Loops are:
    a. Using a control block to keep the app from repeating itself.
    b. Using a control block to change screen colors.
    c. Using a control block to perform repetitive actions based on events in the app.
    d. Using a control block recklessly and without proper supervision.

Summary

In this chapter, we learned how the **Blocks Editor** screen works. Then, we defined **Definition** blocks, wrote in some **Text** blocks, listed items with **List** blocks, added and subtracted **Math** blocks, decided if **Logic** blocks were true or false, used **Control** blocks to loop and other good stuff, colored with **Color** blocks, and then looked briefly into the individual component blocks generated by what we choose in App Inventor **Designer**.

Using blocks, we now better know how to modify and control the components we choose in building apps. But, let's supercharge our apps by adding even more powerful components and concepts. That comes in the next chapter.

4

Mastering Concepts and Advanced Components

Concepts extend the power of our apps (being ways of getting more out of components and blocks), and the various advanced components help us make even more sophisticated apps.

In the latter, the **ActivityStarter** *lets one app call another.* **TinyWebDB** *and* **FusiontablesControl** *let us set up small databases on the web so that apps can access data as needed rather than storing it all in the limited space on a smartphone or other Android device. Bluetooth devices, barcodes, speech recognition, and so on all add power to our App Inventor apps.*

In this chapter, we explore advanced concepts and components and master their use, including:

- Changelog (significant changes in the more recent App Inventor release)
- Concepts (understanding them helps us get more out of App Inventor)
- Advanced components in **Design**'s **Other Stuff** drawer:
  - **ActivityStarter** to interface with other applications and more
  - **BarcodeScanner**, everything from QR codes to cans of tuna
  - Bluetooth wireless communication, client, and server
  - **Notifier**, to put messages on the phone's screen
  - **SpeechRecognizer** knows what we're saying!
  - **TextToSpeech**, or letting Android devices read out loud
  - **Web**, giving us new ways of getting things off the Internet!

- **Not ready for prime time** is a drawer of things not quite finished but still useful:
  - **FusiontablesControl** lets us make use of Google's online (and free) Fusion Tables service
  - **GameClient** and creating games for Android phones
  - **SoundRecorder** and testing one, two, three
  - **TinyWebDB** is all about interfacing with database services on the web
  - **Voting** is a component for running surveys and so forth

Okay, so here's a quick introduction to some quite powerful concepts and advanced components. Then—basics covered—we'll be building real apps in the following chapters and learning components and their blocks in detail!

Changelog

As I've mentioned before, it's still early days in the development of App Inventor with updates (we get them automatically, remember) happening about once a month. Yesterday—June 1, 2011—we got another.

You can tell if there's a new update by the date at the bottom of the **Design** page. This update was dated May 17, but a week or two of testing occurs usually before it gets thrown out to us avid App Inventors.

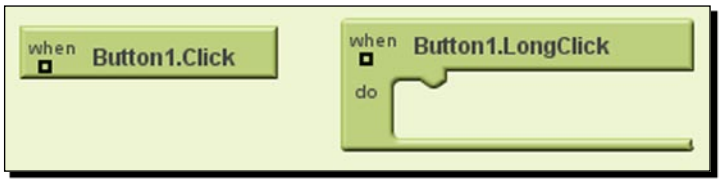
The big new feature is **Web**, a new non-visible component providing communication with websites using HTTP Get and Post requests. It's something a lot of people (me included) have wanted, and we have a section on it later in this chapter showing you what it does.

TinyWebDB (also in this chapter) got promoted from the **Not Ready for Prime Time** drawer in the **Palette** column to **Other Stuff**. Hey, big career move there for that component.

The appearance of framework blocks such as buttons and procedures changed slightly, and buttons got a new block/feature.

Note the small black square beneath **when** in both the blocks below. Left-click on it and the button or other framework collapses as **Button1.Click** is hiding all the blocks in it and saving screen space when not in use. That feature used to be at the bottom of the framework; now you don't have to scroll down 20 or 30 blocks to collapse it.

A new feature for buttons is *long clicks*. Now, buttons work two ways—tap on them and they do one thing, hold them down (long press) and they do something else.



Also, variable definition blocks can be collapsed, and a number of bug fixes were implemented.

More updates on updates as they happen.

Concepts

Concepts are just ways of thinking and doing that improve our creation and troubleshooting of apps, and how we get data into and out of App Inventor. Here's a quick overview of three important concepts.

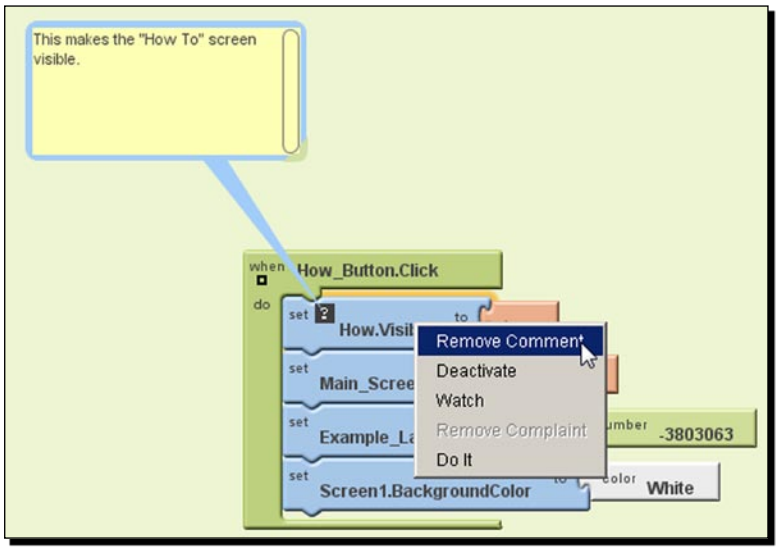
Live development, testing, and debugging

The **Blocks Editor** gives us several features making incremental testing (looking at one part of our apps) and bug fixing easy. Here they are:

Incremental development, meaning we see changes in real time on our phones as we change things in both **Design** and in the **Blocks Editor**. In the following screenshot is an example of this. Right-clicking on a block brings up a menu that:

- Lets us add notes to ourselves and others looking at our source blocks, explaining what it does and why it's there.
- Deactivates the block temporarily as a test to see if it's causing an error.
- **Remove Complaint** (grayed out) removes an error message on the blocks. You'll use this one least of all because App Inventor prevents you from making most errors involving plugging in the wrong block.
- **Watch** lets us watch the value of blocks as the app runs.

- **Do It** allows us to run just that block or just one group of blocks to see if it works correctly.



Specifying sizes of components

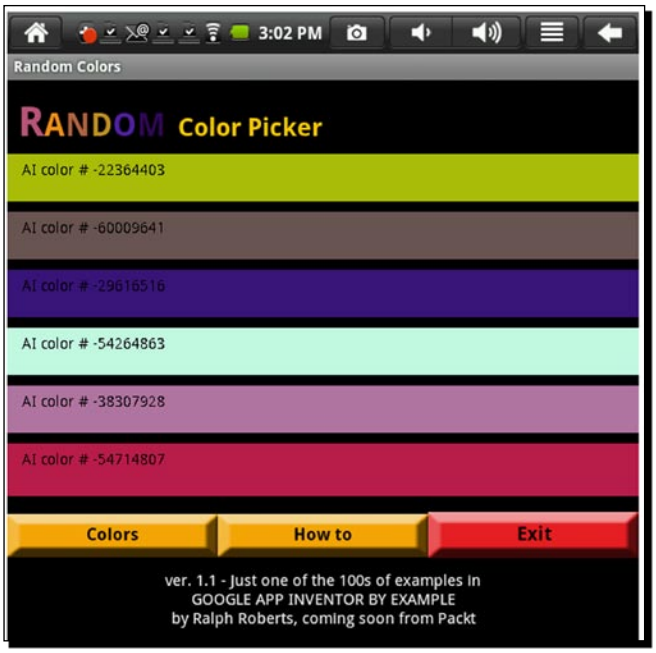
When a visible component is dragged into the virtual phone screen in **Design**, we can size it by auto (let the phone adjust it), by fill parent (let it be as wide as or as high as the screen), or specify a specific width and height in pixels.

Because not all Android devices have the same size screen, it's usually best to avoid putting in a specific width or height. Just test on as many devices as possible or ask others to beta test for you. Sometimes, you just have to guess what will look best on the widest range of phones.

For example, I created a random color picker app (based on what we learned in *Chapter 3*, *Playing with Blocks*) and put it on Android Market. On my Droid 2, used for testing, it looked great.

I downloaded it to my 10.2" tablet computer and the buttons were all off (I'd used specific pixel measurements).

So, I did version 1.1 of the app using a horizontal arrangement with the three buttons inside and all four components set to fill parent. Everything balances, and it looks great and, of course, is a lot bigger on the tablet as shown in the following screenshot:



In short, there are no absolutes in screen design. Hence, it makes designing interesting and fun, I think.

Accessing images and sounds

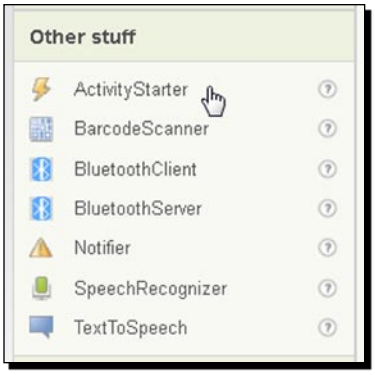
App Inventor apps are able to get media files in three different ways:

1. **Application assets**: files uploaded into **Design** and which are packaged automatically with the finished app. Anyone can use them.
2. **Phone's SD card**: we can play, for example, an MP3 music file from our phone's SD (secure digital) card by using the **player** component and putting in a link like `/sdcard/Music/Blondie/The Best of Blondie/Heart of Glass.mp3`. This would only work on our phone.
3. **Web**: this lets us show or play pictures, music, and video from the web if we know the URL. Must be discrete files, not from a player such as the YouTube videos, for example.

Keep those concepts in mind; they will save you time and trouble. Now, we move on to advanced components.

Other stuff

In the **Palette** column of **Design**—the online part of App Inventor—we find the **Other Stuff** drawer (see the following screenshot) toward the bottom:



Let's go through these, starting with the most powerful component in all of App Inventor land: the **ActivityStarter**.

ActivityStarter

Both Google and, now, myself have made a big thing of how App Inventor insulates you from having to know and use all that esoteric, confusing Java code (*shudder*).

The Glorious Googlelab Geeks (and I mean that as a very positive compliment) have gone to great lengths and continue to do so in developing this "shuffle around a few blocks" app-building playhouse for us.

However, there are still a very few "holes" in the floor—places where it is still necessary to occasionally reach down and grasp a handful of code (just wash afterwards, eh?).

The **ActivityStarter** is perhaps the most complex component—due to having to type in actual Java code in **Design** to set it up. Drag an **ActivityStarter** component onto the virtual screen, and look in the **Properties** column. There are several blocks such as **Action**, **ActivityClass**, **DataType**, and so on. Yeah, Java stuff. Don't worry, I'll give you a good source to find those, and you won't even have to learn Java.

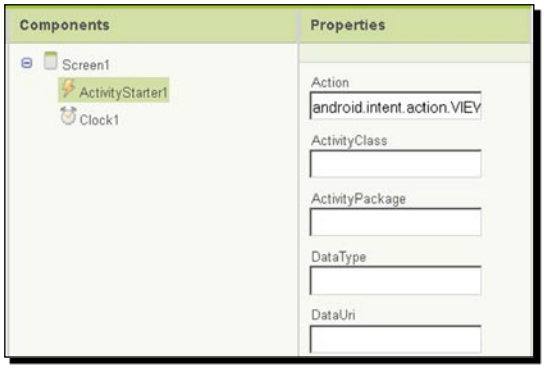
We use **ActivityStarter** to begin things that occur **outside** our apps—load a webpage, send e-mail, open up the phone's built-in calculator, and much more.

As a fast example, let's do the simplest activity: opening up the web browser on the phone and loading a specific page. This is also one of the shortest and easiest apps you can build; it's called a bookmark app.

Time for action – building bookmark apps

Let's create an icon on the phone that, when tapped, opens a webpage.

1. Pull an ActivityStarter component onto the virtual phone screen in **Design**.
2. Pull in a **Clock** component (it's in the Basic drawer). This component lets us end the app after the page is called.
3. In the **Action** field, type `android.intent.action.VIEW` ('VIEW' is in all caps, see below).

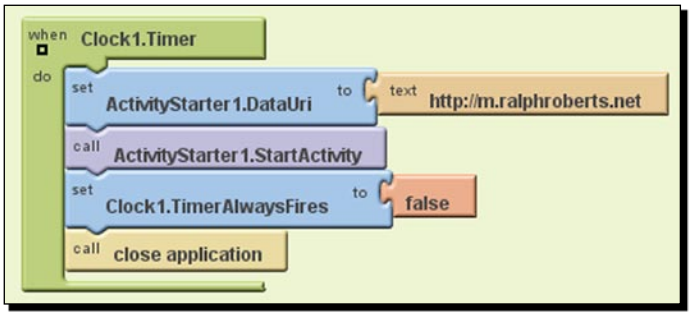


4. In the **Properties** column for **Screen1**, add an icon if you like.
5. In the **My Blocks** column of the **Blocks Editor**, we get the **ClockTimer1** framework (automatically created for us).
6. From the **ActivityStarter1** drawer (more *automagic* from App Inventor) comes the set **ActivityStarter1.DataUri** to block and plug it into the clock timer framework.

7. Use just a regular text block to add the web address (URL). In the following illustration, I'm using that for the mobile version of my personal blog (y'all come and visit).

8. A call **ActivityStarter1.StartActivity** block kicks off the activity (loading in the web page designated in the first block).

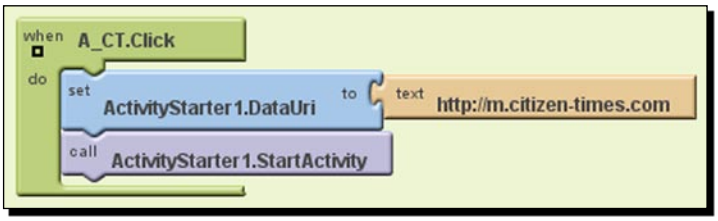
9. The clock timer does not fire (false), which causes this group of blocks not to be repeated, and the **close application** block (from **Built-In**/**Logic**) exits the app.



What just happened?

We now have a bookmark app.

But, what if you want to keep people in your app instead of just opening a web browser on their phone (that is, giving up control of their session)? In my *AVLnews* aggregator (collection) app, I want just that. So, I use a button, as below, and let users choose from three screens of news sites. When they hit the hardware back button on their phone, they're in my app again.



ActivityStarter gets a little more complicated with other activities. Here's how we call the calculator on the user's phone. Fill in these fields in the **Properties** column of **Design** for an **ActivityStarter** component:

- **Action**: `android.intent.action.MAIN`
- **ActivityClass**: `com.android.calculator2.Calculator`
- **ActivityPackage**: `com.android.calculator2`

To use it, go to the **Blocks Editor** and build a button as follows:

Tapping the button brings up the calculator. When the user hits the back button to exit the calculator, he or she is returned to your app, now in progress.

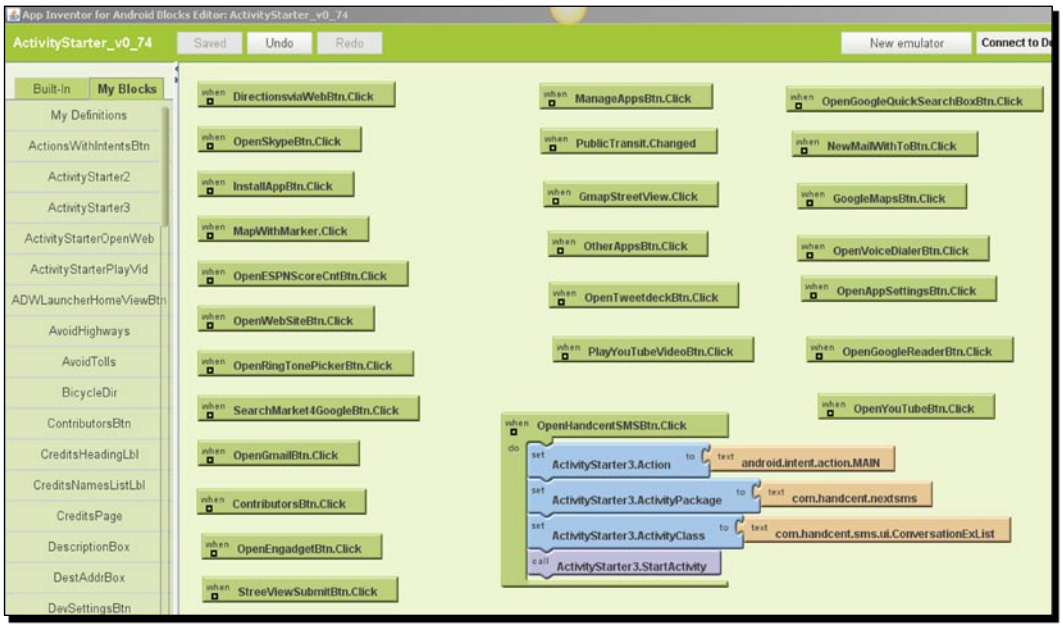
We'll create some apps using **ActivityStarter** beginning in *Chapter 5*, *Apps That Communicate*, for e-mailing and other practical, useful communications tasks.

But... oh, yeah... I hear you asking, "Where do you find those convoluted Java statements to use in **ActivityStarter**?"

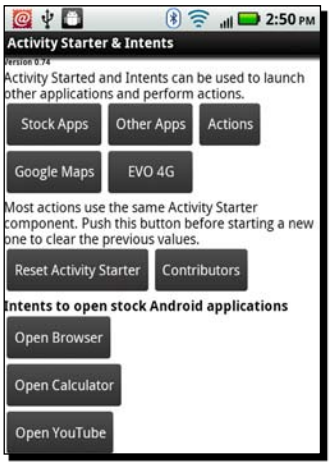
Best reference source I've found so far is *Using the Activity Starter* by *Gene Kupfer* and *Tony Barnes*. It's an App Inventor app you can download at `http://www.taiic.com/downloads/` for free. Get the latest version (at the time of writing): `ActivityStarter_v0_74.zip`. Also some other good stuff there. Gene and Tony are two of the good guys in helping people with App Inventor on the App Inventor forums (`http://appinventor.googlelabs.com/forum/`).

Once you've downloaded the ZIP file to your computer, upload it to your **My Projects**, as we learned how to do in *Chapter 2*, *Learning Components* (left-click on **More Actions**, left-click on **Upload Source**).

Once it's uploaded to **My Projects**, it opens up in **Design**. Go to **Blocks Editor**, and you'll see a bunch of collapsed example buttons as shown in the following screenshot. Click on **Connect to Device**, select your phone, and you can actually test these many activity starter examples in real time!



To get the parameters you need for a specific activity starter action, expand that button (as in the `HandcentSMS` action previously (requires `HandcentSMS` be installed on your phone). There you have the action, activity class, and activity package information you need. The following is what Gene's sample app looks like on my phone:


2:50 PM
Activity Starter & Intents
Version 0.74
Activity Started and Intents can be used to launch other applications and perform actions.
Stock Apps
Other Apps
Actions
Google Maps
EVO 4G
Most actions use the same Activity Starter component. Push this button before starting a new one to clear the previous values.
Reset Activity Starter
Contributors
Intents to open stock Android applications
Open Browser
Open Calculator
Open YouTube

Gene, by the way, is also the developer of **Marketizer**, a free app for putting our App Inventor apps on Android Market. I find it indispensible. Thanks, Gene!

BarcodeScanner

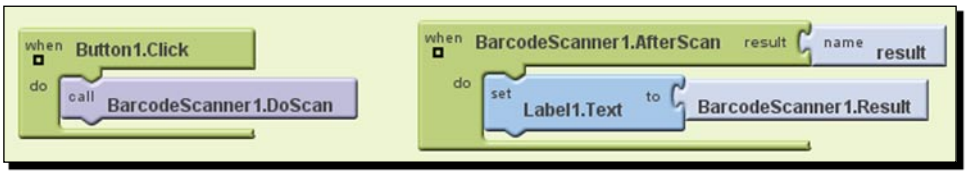
This powerful component is incredibly trivial to program and use. It, of course, scans barcodes. Barcodes cover everything from the Bookland/EAN codes on the back of books, UPC commercial barcodes on toothpaste, cans of beans, and almost every product you can think of, and many other kinds.

`BarcodeScanner` reads many of these, including those QR codes (the square things with black squiggles on them that look sort of like a maze). We see those everywhere from websites to billboards now. They are often used to provide links to download apps.

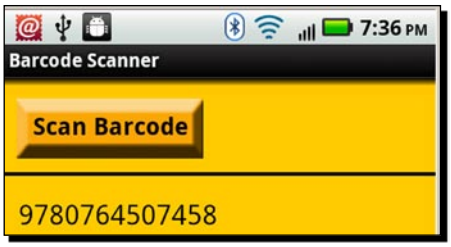
A prerequisite for using this component is that you have the ZXing Project's Barcode Scanner (free, get it on the Android Market) installed on your Android device. Now, let's build a barcode scanner.

Time for action – our very own barcode scanner

1. In **Design**, you need a button and a label. Format them using horizontal arrangements just for practice.
2. Drop in a **Barcode Scanner** component from the **Other Stuff** drawer.
3. Switch over to the **Blocks Editor** and drag out your button's click framework (as shown in the following illustration). Pull **the BarcodeScanner1.DoScan** into the framework.
4. Get a **BarcodeScanner1.AfterScan** block and add the **Label1.Text** into it, and click the **BarcodeScannerResult1** block into that.



Now, find some barcodes and scan them so you can see the results. Here's what I got when I scanned one of my books from the bookshelf (*Rebol for Dummies*). The result is the ISBN number (how books are tracked in bookstore inventories).



What just happened?

We now—with a very little work—have an app that scans barcodes.

Speaking of barcodes, here's an example of a *QR barcode* (which can contain up to 250 characters). Scanning this one on your phone will also give you the link to the mobile version of my blog. Try the app above on it.

Bluetooth

Bluetooth (as defined in the Wikipedia) is a proprietary wireless technology standard for exchanging data over short distances (using short wavelength radio transmissions) from fixed and mobile devices.

Most of us with Android phones know about wireless earpieces that let us talk safely while driving, wireless keyboards, and so forth. These devices communicate with our phones using Bluetooth technology.

App Inventor provides two Bluetooth components in **Other Stuff—BluetoothClient** and **BluetoothServer**. Although definitions are sometimes blurred in actual practice (overlap), basically a *client* accesses a server (like your e-mail program gets e-mail from a mail server), and a *server* provides a service to one or more clients.

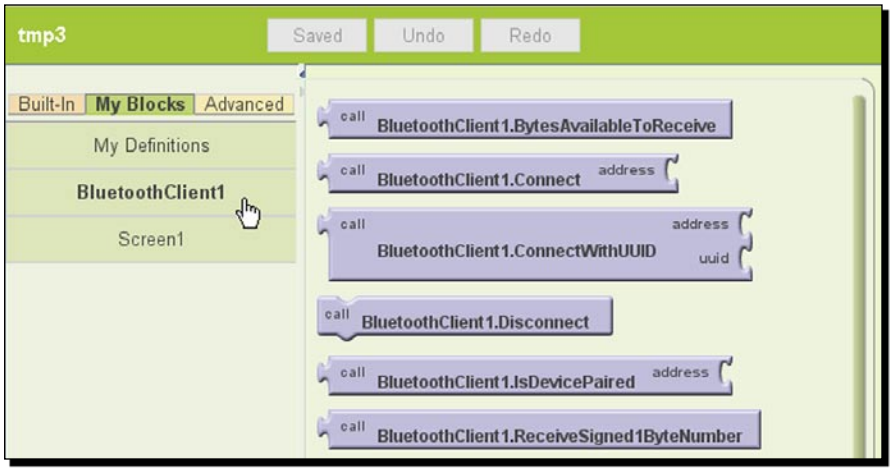
Bluetooth uses profiles to connect with various devices. There is something close to 30 of these now available. Hands Free Profile (HFP) and Headset Profile (HSP), for example, are both used in pairing and using Bluetooth earpieces with your phone. Video Distribution Profile (VDP) is for streaming video signals and so forth.

App Inventor's current implementation of Bluetooth—the client and server components mentioned previously—only supports Serial Port Profile (SPP), also referred to as RFCOMM. SPP is actually the most common Bluetooth profile, often used on controllers and in communicating with instrumentation and similar devices.

So, until the developers add more features to App Inventor's Bluetooth components, it's a bit limited for most of us.

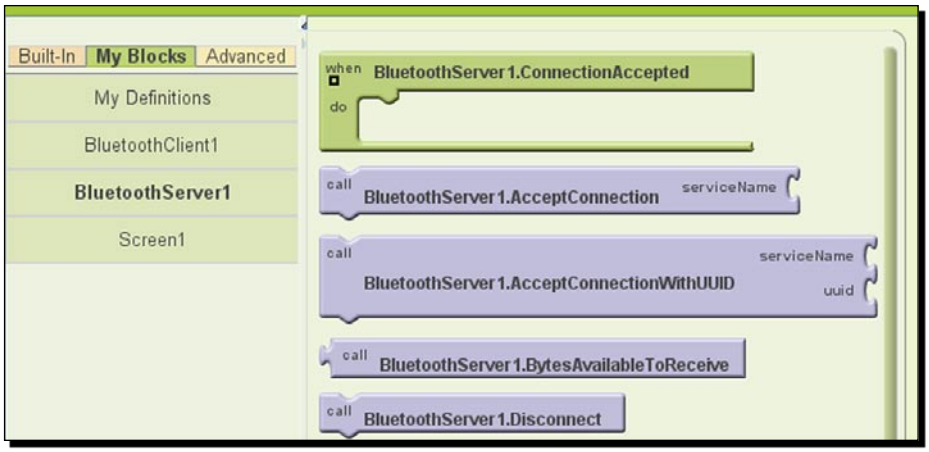
When we drag-and-drop a **BluetoothClient** component on the virtual phone screen in **Design**, a drawer is automatically created in **Blocks Editor** (in the following illustration, we have numbered it as **1** because it's the first one—we can have more than one Bluetooth client).

Only some of the available blocks are shown in the following screenshot:

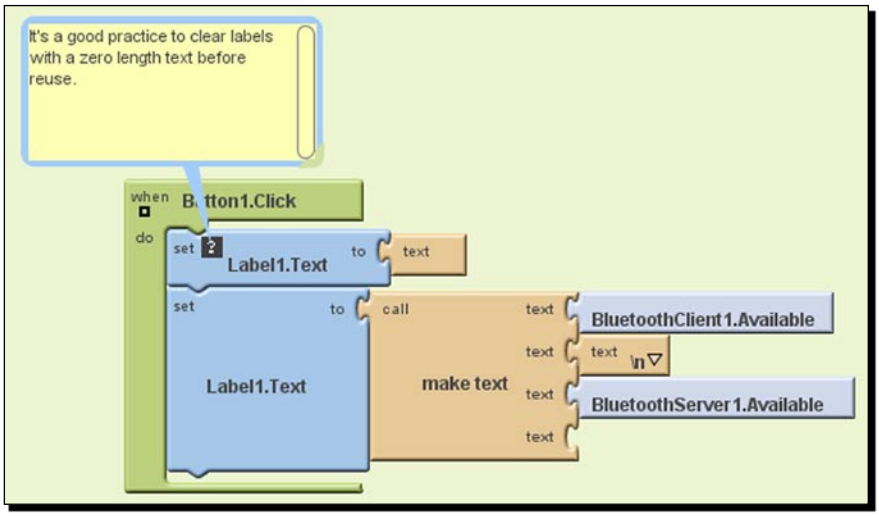


Download from Wow! eBook <www.wowebook.com>

And, (again because operations sometimes blur or overlap) the blocks created for the **BluetoothServer** component (partial drawer shown in the following illustration) are somewhat similar to those in the client drawer.

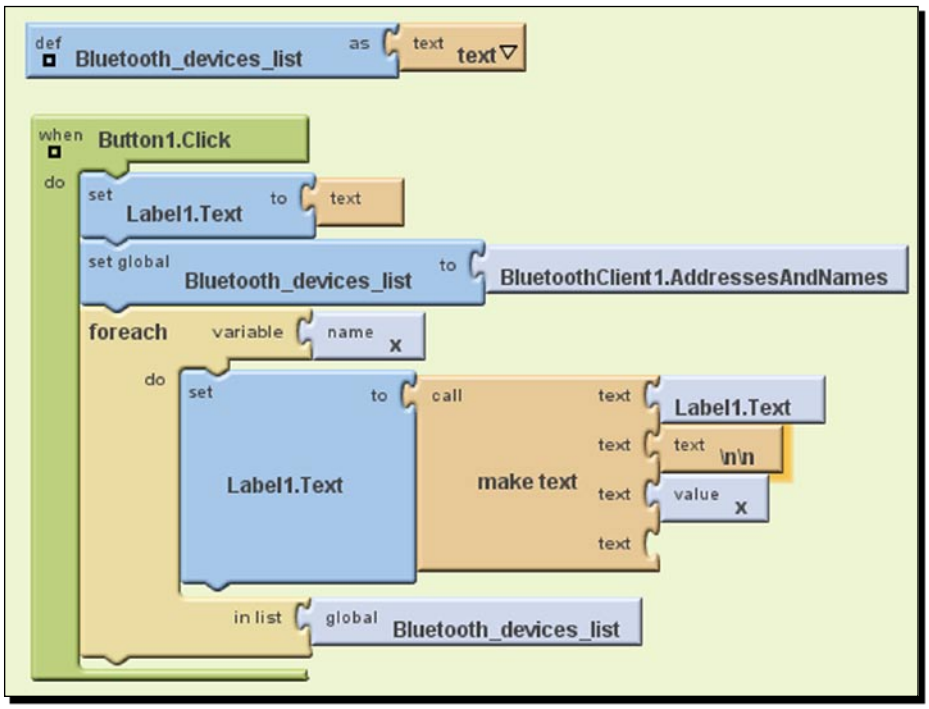


First question we should ask is: "do we have active Bluetooth on our phones?". The following is a quick test for both client and server:



And, checking the results on my phone (see the following illustration), I find I do. Which means Bluetooth is activated in **Settings** on the phone (or other device).

Now, are there devices set up (called pairing) on my phone? Well, easy enough to check. The **BluetoothClient1.AddressesAndNames** block returns a list of those devices. We learned how to display a list in *Chapter 3*, *Playing with Blocks*, so we check like this:



Looking on my phone, I find three devices paired and their addresses (shown in the next screenshot). I have a Bluetooth keyboard (very handy for those long e-mails or even a bit of writing while waiting in the dentist's office) and two earpieces. Love that Voyager PRO+ and use it all the time.

But, please remember, even though App Inventor sees these devices, unless they have SPP protocols, no connection can be made (and none of these devices do).

An example of an SPP device that we can write App Inventor apps for using the components in this section is the Bluetooth of the *Arduino* board.

Arduino is an open source electronics prototyping platform with simple hardware and software. You can hook various sensors to it to monitor the environment (temperature, wind, humidity for a weather station, for example) and/or control motors, lights, and other devices.

These Bluetooth components can also be used in conjunction with the **Lego® Mindstorm®** components in controlling robots. I suspect someone on the Google App Inventor team works with robots or plays with them as a hobby. Sounds like real fun.

But, while limited now, we'll keep our eye on AI and Bluetooth—some great potential there.

Notifier

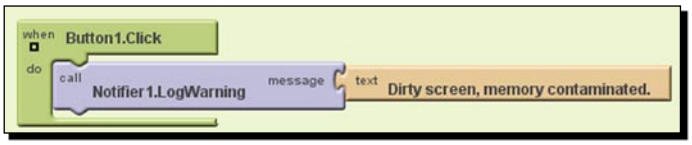
Now, it's time to play with the **Notifier** component, which can be both fun and useful. As the name indicates, *notifiers* are a way apps get your attention.

Some of the notifiers you don't even see on the phone's screen. Drop a **Notifier** onto the virtual phone screen in **Design**, open **Blocks Editor**, and look in the **Notifier1** drawer.

There are a couple of frameworks at the top—one for after choosing a notifier (what happens after a notifier is activated) and one for acting on text input from a notifier. We'll get to those in a moment.

The next three blocks—**Notifier1.LogError** (or **2**, or **3**, or however many you use in the app), **Notifier1.LogInfo**, and **Notifier1.LogWarning**—do not show on the phone screen. Instead, they make entries into your phone's log and are useful in debugging apps.

We'll use a button (as below) to activate a **Notifier1.LogWarning**, which creates a warning line in our device's log.

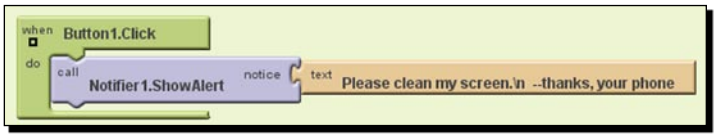


Okay, reading the log falls outside the scope of this book, but can be done with the Android SDK (Software Development Kit). See how to install in *Chapter 3*, *Playing with Blocks*. But, it's there if you want to learn it. You can also do captures of your phone's screen with it and load other emulators such as one for a tablet computer (I showed you that in *Chapter 1*, *Obtaining and Installing Google App Inventor*).

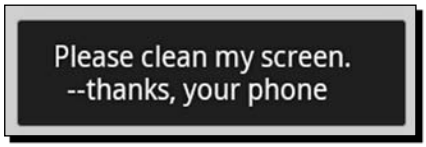
Clicking the button twice causes these two warning lines to show in the phone's log as here:

Log
Time
pid
tag
Message
06-07 19:44:... D 1238 WifiSe... releaseWifiLockLocked: WifiLock{NetworkLocationProvider type=2 binder=android.os.Binder@45ed7420}
06-07 19:44:... W 16... Notifier Dirty screen, memory contaminated.
06-07 19:44:... W 16... Notifier Dirty screen, memory contaminated.
06-07 19:44:... D 1337 Polling... wake lock acquired
06-07 19:44:... D 1337 Polling... decideWhoNeedsPolling(): polling PushService

The **Notifier1.ShowAlert** block causes a message to appear on the phone's screen. It lasts for a few seconds and fades. This might be considered a polite way of imparting a warning or requesting an action. The following is one way to activate it, using a button:

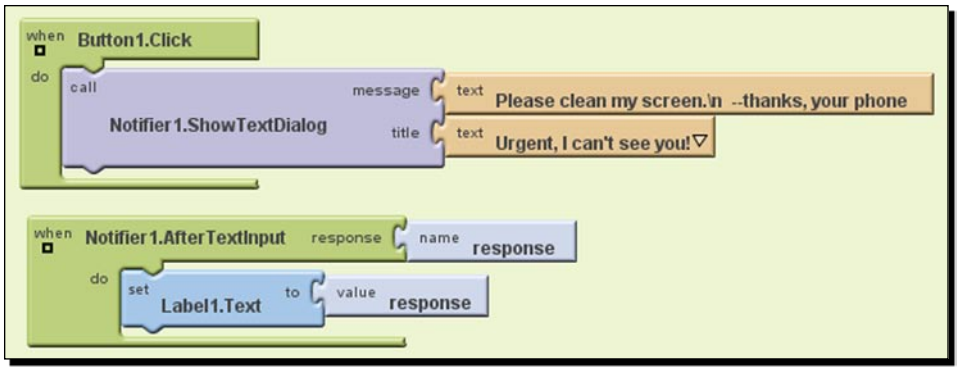


And, it looks like this when displayed:

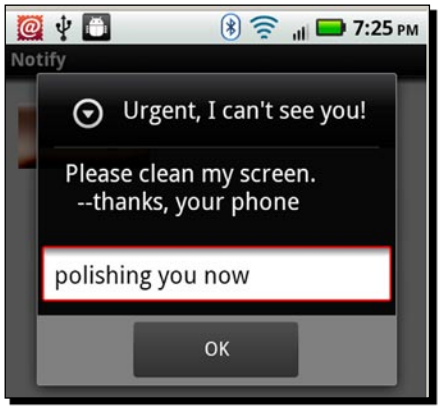


Now, moving along, what if we should want the app to be a bit more "in your face" to the user? Using the **Notifier1.ShowTextDialog** allows us to have a more ... er... *noticeable* notifier on the screen.

Also, by pulling out a **Notifier1.After TextInput**, we can set up the app to react to whatever the user might type in the *integral* (built-in) textbox in this notifier. We arrange our blocks as follows:

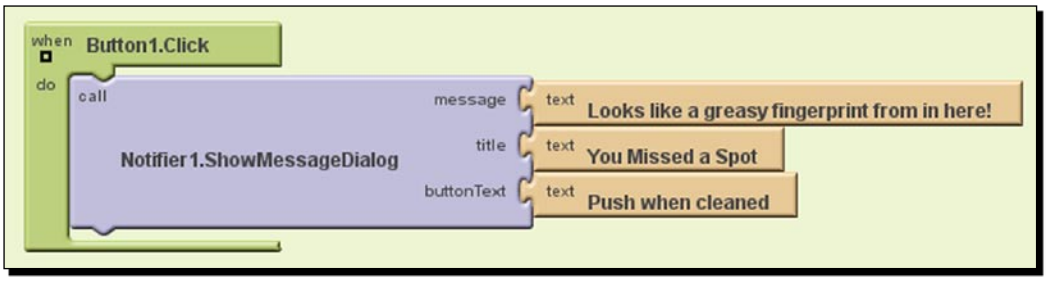


Tapping the button on the phone's screen overlays a definitely distinctive message on the phone, as we can see (I mean, *really* see) here:



The user types in a message, taps the integral **OK** button, and we display it here in a label. In a real-world production app, we would have the app go forward (it's called *branching* in programming) depending on what was typed in. Here's how our little sample notifier displays the text:

The **Notifier1.ShowMessageDialog** gives us that "noticeable" notifier on the screen, but adds an integral button. You put the parameters in it as shown here:



The result looks as the following screenshot. The notification stays on the screen until the button is pushed.

Neat, huh? Notifiers are a very useful tool in apps and add to their professional appearance and operation.

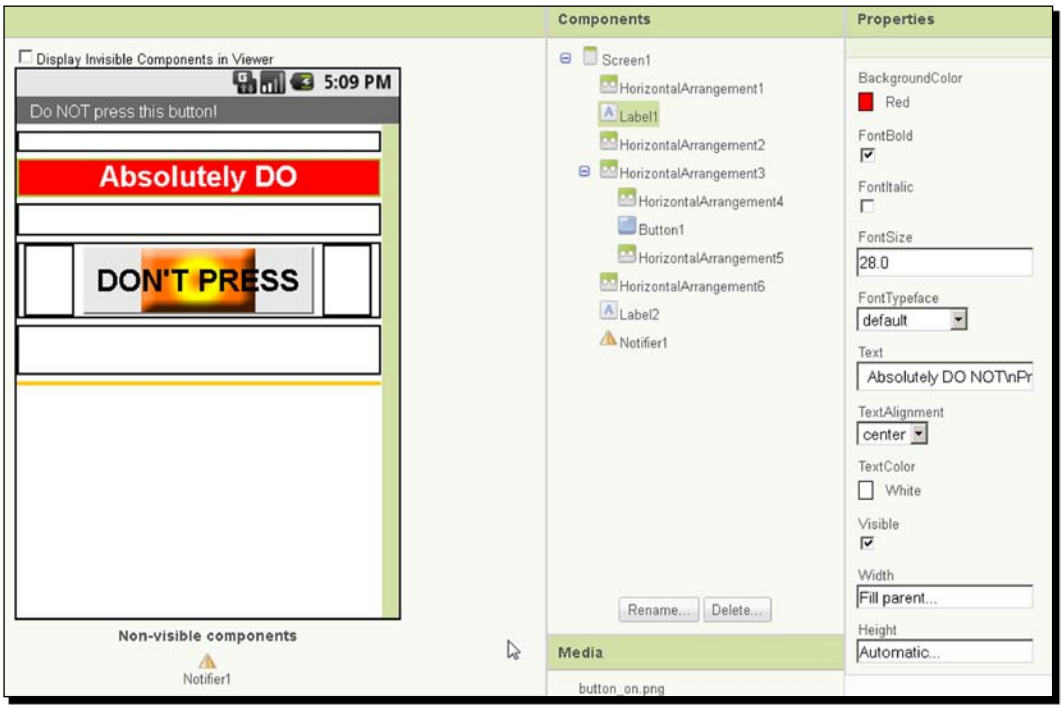
And, I saved one block for you to have some fun with!

Have a go hero – notifier with two buttons

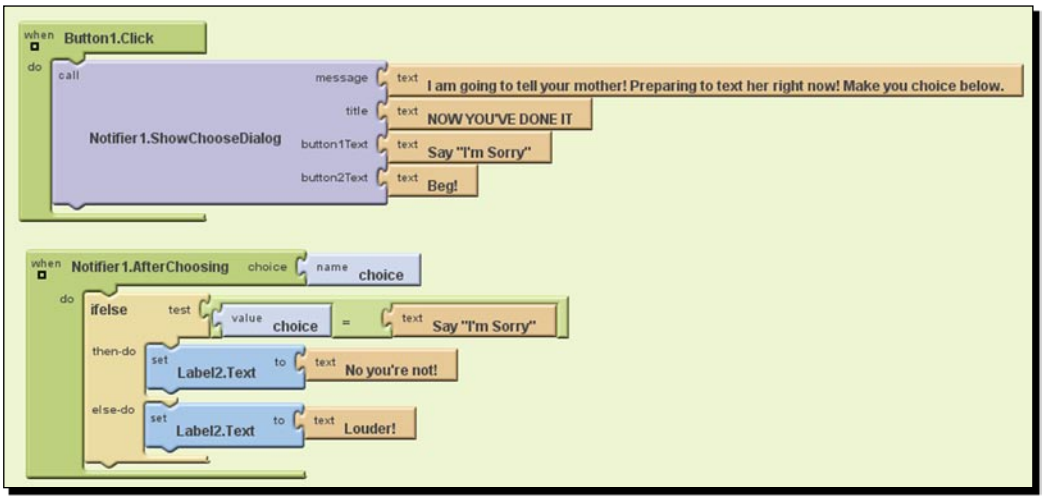
Construct this humorous (well, kind of) use of the **NotifierShowChooseDialog** notifier block. It's also good formatting practice for you.

1. You want to get a look like the following screenshot. You'll need a label, a button, and a second label.

2. As shown in the next screenshot, you'll want to use **HorizontalArrangement** component for spacing and centering. You don't have to make it exactly like mine, but you should always strive to make it neat.



3. In the Blocks Editor, you'll only need the **Button1.Click** and **Notifier1.AfterChoosing** block frameworks, as shown in the following screenshot. Fill them as shown here:



Now you've got the fun app that shows how sassy your phone can get. Pressing the **DON'T PRESS** button brings up this notification:

Pressing the **Beg!** button returns "Louder", and the **Say "I'm Sorry"** button gives the following screen:

Package that, download it to your phone, and you'll be able to show your friends that sometimes your phone has an attitude.

Okay, now (especially after some of the answers it gave us previously) we need to have a little talk with our phones.

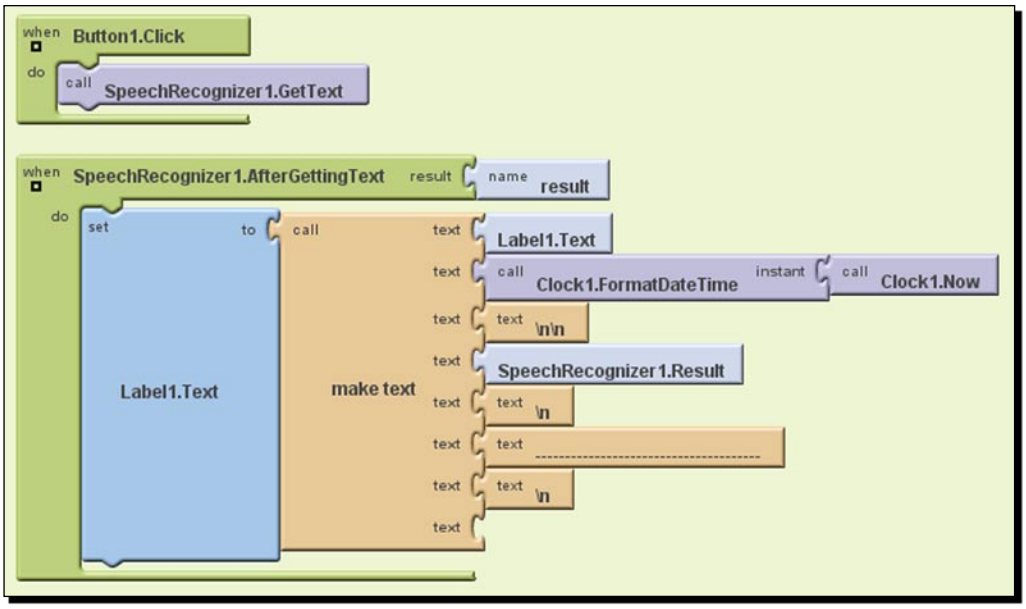
SpeechRecognizer

The **SpeechRecognizer** component allows your phone to understand what you say. And, its use is simple.

Time for action – build a time- and date-stamped Voice Note Taker

1. In **Design**, you need a button and a label. Make the label fill parent in width but automatic in height so that it can scroll. As ever, use horizontal arrangements for spacing.
2. Drop in a **SpeechRecognizer** (from the **Other Stuff** drawer) component and a **Clock** component (to provide time stamping; it's in the **Basic** drawer).
3. In the **Blocks Editor**, pull out the **Button.Click1** framework (shown in the next screenshot), and put the **SpeechRecognizer1.GetText** block into it (automatically created when you drop this component onto the virtual phone screen when designing the app). We get the framework from the same drawer for holding blocks that specify what happens after speech recognition is called.
4. Drag out the **SpeechRecognizer1.AfterGettingText** framework from the **SpeechRecognizer1** drawer. We'll build our display of each voice note in it.

5. Get **Label1.Text** from the **My Blocks/Label1**. Use a make text block (**Built-In/Text** drawer). Basically, we join new notes to the existing text, automatically adding the new note's date and time, putting in carriage returns (line breaks or `\n`).



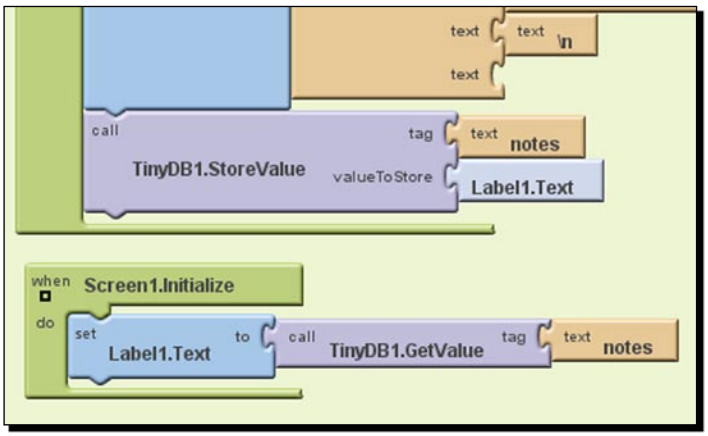
The result should look something like the following:

10:34 PM
Voice Note Taker
Listen to Your Master's Voice
Jun 7, 2011 10:32:50 PM
mary had a little lamb
Jun 7, 2011 10:33:00 PM
roses are red
Jun 7, 2011 10:33:09 PM
violets are blue
Jun 7, 2011 10:33:20 PM
pick up milk and bread
Jun 7, 2011 10:33:35 PM
finish reading this book

What just happened?

With only a few blocks and components, you built a pretty powerful Voice Note Taker where you can verbally enter reminders, ideas, grocery items, and so forth.

To turn this exercise into a useful real-world program, you need persistent data—that is, the notes stay available even when you exit and re-enter the app. Use the **TinyDB** component (Basic drawer) for persistency. Just store the value of **Label1.Text** (as shown in the next screenshot) and add a **Screen1.Initialize** block to recall the contents of **Label1.Text** when the app starts up again. We'll revisit this app later and add a few other refinements to make it worthy of publishing on the Android Market.



This app uses the built-in speech recognition in your Android device, which in turn relies on software out in the Google cloud. Meaning, an Internet connection via Wi-Fi is required (and the speech recognizer will attempt to get one if it's not already there).

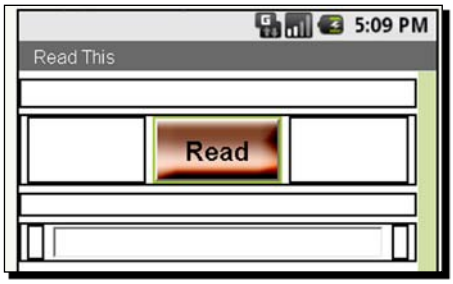
So, we can speak to our phones and they understand, but what about teaching them to read aloud? Let's do that.

TextToSpeech

App Inventor enables Android devices to read quite well.

Time for action – reading aloud

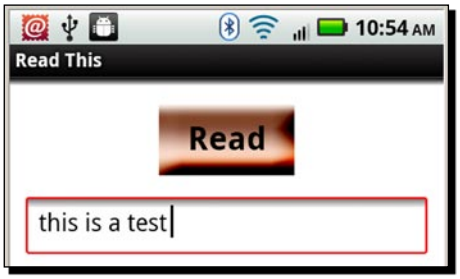
1. Set up a way to input text to be read aloud in **Design**—all you need is a button, a **TextBox** (in the **Basic** drawer) component, and to drop in a **TextToSpeech** component (it gives us a place to enter text to be read). On the virtual phone screen shown in the next screenshot, the other components are just horizontal arrangements for spacing and centering the button.



2. Moving to the **Blocks Editor**, here's all we need—a button framework (get it in the **Button1** drawer in My Blocks) and the **TextToSpeech1.Speak** block from the **TextToSpeech1** drawer, also in **My Blocks**.

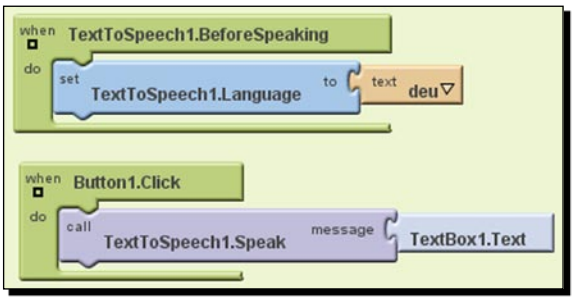
What just happened?

And that's it for setting up the reading aloud app. Go to your test device's screen, tap the textbox to bring up a keyboard, and enter text (as shown in the next screenshot). Press the read button to listen.



Download from Wow! eBook <www.wowebook.com>

What about other languages than English? Well, you can have some fun with accents by using the **TextToSpeech1.BeforeSpeaking** framework block and the **TextToSpeech1.Language** block, as shown in the following screenshot. This will read your text in a German accent. Put in `fra` and it will use a French accent.



Get other language and country codes at: `http://appinventor.googlelabs.com/learn/reference/components/other.html#TextToSpeech`.

Use a service such as Google Translate; drop the translated text into text-to-speech, and it will read the translated language in the accent you have set. So, German sounds German.

Using the Google Translate API (application programming interface), it would be easy to pass the text you type in to Translate, get it back, and have your app read it. I've seen it done using the **TinyWebDB** component, but it would probably be even easier with the new **Web** component.

Alas, the problem with that lies in Google's announced cancelling of the Translate API soon, due to abuse. It's possible, according to an article I just saw, that it will continue it as a paid service. So, we need to find some other online translation software that would allow third-party applications connecting to it.

Anyway, have fun with text-to-speech, and speaking of **TinyWebDB**...

TinyWebDB

The **TinyWebDB** component is now in the **Other Stuff** drawer in the **Palette** column of **Design**, having recently gotten promoted from the **Not Ready for Prime Time** drawer.

Remember the example just before of using **TinyDB** to store persistent data on your phone from app session to app session? Well, **TinyWebDB** does the exact same thing, only the values are stored on the web (you'll need a TinyDB web service out in the cloud to take advantage of this).

For testing purposes, when you drop a **TinyWebDB** component (and, as ever, you can have more than one) onto the virtual phone screen in **Design**, a link is automatically configured to an App Inventor TinyDB test site (look in the **Properties** column, the link (**ServiceURL**) is `http://appinvtinywebdb.appspot.com`). In *Chapter 6*, *Apps That Remember*, I'll show you how to set up your own service.

The reason for having your own service is because the test site is limited in capacity, and your data is not guaranteed to stay there for very long.

TinyWebDB functions—unlike most things in **Blocks Editor**—will not work on your test phone. You have to package and download the app to the phone, and then run it.

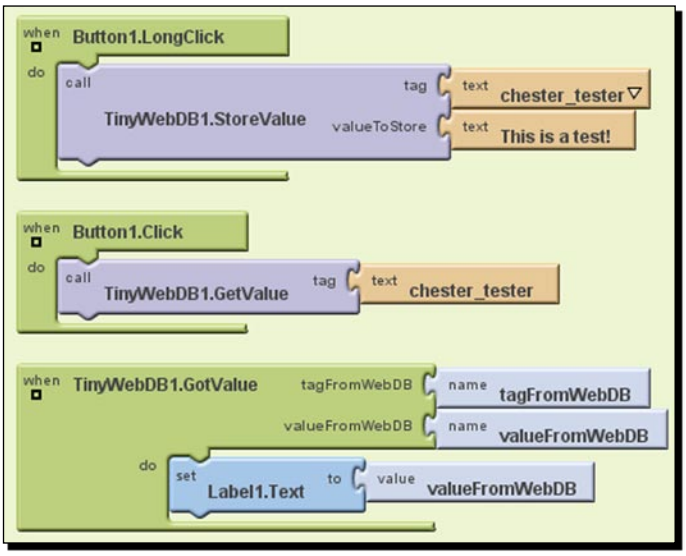
Time for action – storing persistent data on the web

Okay, in **Design**, all you need is a button, a label, and the non-visible **TinyWebDB** component. We now move to the **Blocks Editor**.

We will be using one button for two functions (thanks to the recently added long press feature).

1. First, drag out the **Button1.LongClick** framework (shown in the next screenshot), and use that to store a value in the test web service with the **TinyWebDB1.StoreValue** block from the **TinyWebDB1** drawer on the **My Blocks** tab (again, the URL is already configured in **Design**). As with **TinyDB**, you use a **tag** (name of the stored value) and the value itself in **text** blocks clicked in as shown.

2. Next, set up the **Button1.Click** framework (a short tap to the button sets it off) to retrieve our stored value, calling it by its name (tag).



3. Finally—in the previous bottom block group (**TinyWebDB1.GotValue**), we set up how the retrieved value is handled; that is, sent to the label on our phone.

4. Install the app on your phone. Give a long press to the button (it should vibrate a short burst when done). Tap it then, and the label shows the text value (shown in the following screenshot) pulled off the web. It works!

What just happened?

Simple storage of values for data persistence is markedly simple once you have a reliable **TinyDB** service. For more complex data applications—albeit Shival Wolf from the App Inventor forums has done some amazing work interfacing with PHP and MySQL databases—it gets a bit complicated.

Luckily, we now have the recently introduced Web component, so here's how it works.

Web

Web (newly added to the **Other Stuff** drawer) is another non-visible component that provides functions for HTTP GET and POST requests—GET and POST being methods of sending and receiving data from websites.

"Ah, yes," you say with a nod, "but didn't **TinyWebDB** exchange data with websites?". Yes, and so does **FusiontablesControl** coming up later in the chapter. But, of the three, **Web** is the easiest to use and, in my opinion, potentially at least the most powerful.

TinyWebDB, as it was designed to do, excels in providing persistent data storage between sessions of apps, doing it on the web as opposed to **TinyDB** just on the one phone. TWDB, as my friend Roger on the AI forums shortens it, is quite useful in that respect. But, ask it to interface with existing large databases and that gets a little hairy.

In the web world, LAMP rules—LAMP being an acronym meaning Linux (operating system), Apache (web server software), MySQL (powerful relational database), and PHP (the most widely used scripting language on the internet). You find this combination everywhere.

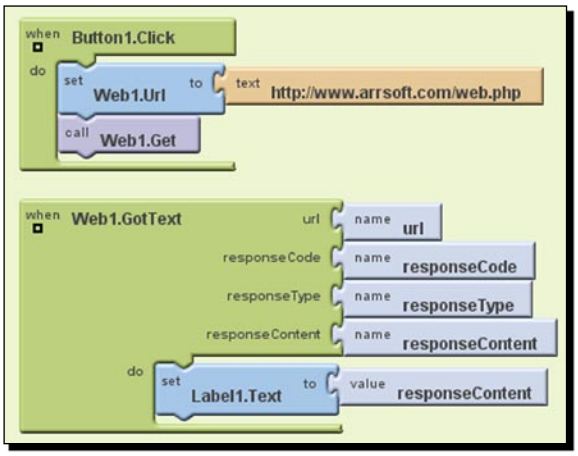
It would be nice to be able to interact with LAMP from our phones, it being so ubiquitous (a fancy way of saying everywhere) and all. Ah ha! That's what **Web** does!

One app I've been wanting to do is called *Galactic Headlines*—fanciful and humorous news leads from all over the universe. I've written (and published) this program in several other computer languages, but the database is so large I knew it would not fit easily on phones with their limited memory.

I tried **TinyWebDB**, but that was getting convoluted. Fusion Tables were slow getting data to the phone, and you had to make the tables public. Then, just a little over a week ago as I write, AI was updated and the Web feature added, which solved my problems. Galactic Headlines (generates trillions of possible headlines) is already on the Android Market. Download it for free (searching for `arrsoft` will find all my published apps for you, so enjoy).

Here's how simple the technique is (albeit access to a web server is required).

Only a button and a label are needed. The **Web** blocks (shown in the next screenshot) get and display the headlines.



A simple PHP script on the server pulls random parts of the headlines out of a MySQL database.

```
<?php

include ('/home/arrsoft/php.conf');

mysql_connect(localhost,$name,$pass);

@mysql_select_db($database) or die( "Unable to select database");

$nouns =  mysql_query("select * from nouns;");

$random_row = mysql_fetch_row(mysql_query("select * from nouns order by rand() limit 1"));

echo $random_row[1];

mysql_close();

?>
```

And, they are displayed on your phone (this screen is from the published app).

10:45 PM
Galactic News Service
Galactic Headlines
MAUVE ONE ARMED ROBOT GOOD-NATUREDLY ELOPES WITH YOUTHFUL SOAP OPERA STAR
DETERMINED SENTIENT GALAXY STANDOFFISHLY FINDS SLOPPY RETIRED MILITARY MAN
COMPLAINING JAR OF LINDENBERRY JELLY DELIGHTFULLY TAKES LUNCH TO OPPORTUNISTIC SMALL ELEPHANT
LIGHTSLATEGRAY-COLORED LADY IMPARTIALLY YANKS NOSE OF DISCREET RANCHER
GUILTY TROLL FORESIGHTEDLY HUMORS STUCK UP CONFESSION WRITER
Headlines
About
Exit
ver. 1.0 by Ralph Roberts, ©2011 ArrSoft, arrsoft.com

We'll look at passing SQL queries to databases (MySQL, PostgreSQL, and so on) and more in *Chapter 6*, *Apps That Remember*.

And, that completes the **Other Stuff** drawer. Now for the **Not Ready for Prime Time** players.

Not Ready for Prime Time

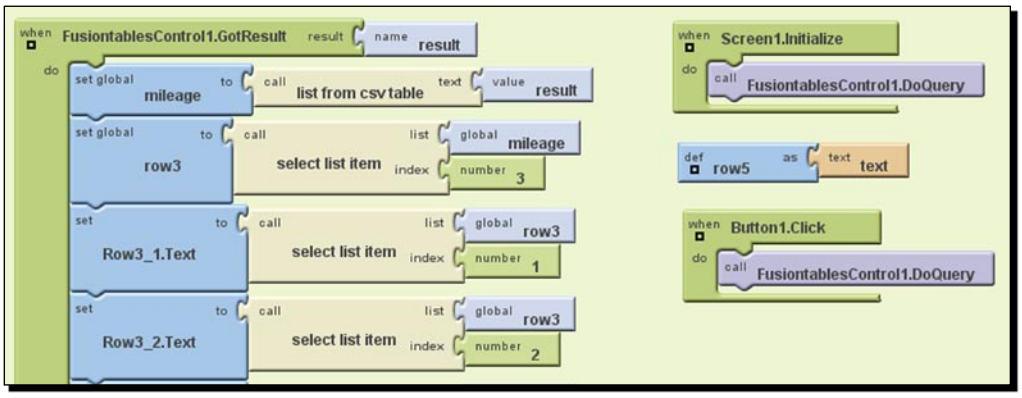
In this drawer, we find four components that, while usable, are not quite in a finished state yet. Still, we'll make good use of them.

FusiontablesControl

Fusion Tables are a free Google service (see `http://www.google.com/fusiontables/public/tour/index.html` for a good overview).

Fusion Tables are, in short, a visual database, albeit more akin to a spreadsheet than "real" databases like the aforementioned MySQL and PostgreSQL or the big commercial one, Oracle. But, if you do not have access to your own server, this might be something very appealing to add a powerful web dimension to your apps.

As I related earlier in the book, I have an app underway using Fusion Tables to keep track of vehicle mileage for one of our companies. It's trivial to get the data in and out of Fusion Tables via the web. Getting it into a readable format (as shown in the next screenshot) is a bit more tedious.


when FusiontablesControl1.GotResult result name result
do set global mileage to call list from csv table text value result
set global row3 to call select list item list global mileage index number 3
set Row3_1.Text to call select list item list global row3 index number 1
set Row3_2.Text to call select list item list global row3 index number 2
when Screen1.Initialize
do call FusiontablesControl1.DoQuery
def row5 as text text
when Button1.Click
do call FusiontablesControl1.DoQuery

But, it looks good on the phone, as seen here again:

Vehicle	Start	Stop	Type	Date	Miles
Kia	30000	35000	Personal	4/26/11	500
Target	68000	69000	Notary	4/30/11	1000
Trumpet	42000	43000	Notary	4/29/11	1000

However, now that **Web** has come along, I'm switching the app over to that and putting the data on one of our company servers.

We will look more in depth at Fusion Table apps in *Chapter 6*, *Apps That Remember*, and do one so you'll be able to add them to your toolbox. And, yes, I'll show you how I formatted that table on the screen previously.

GameClient

GameClient is still under development, but is meant to (and does to some extent now) communicate with online game servers for multiplayer games. It lets you develop clients for games you host on servers such as Google's App Engine (see `http://code.google.com/appengine/docs/whatisgoogleappengine.html` for details on that), or on your own game server.

Chapter 9, *Games and Animation!* is where we get into games.

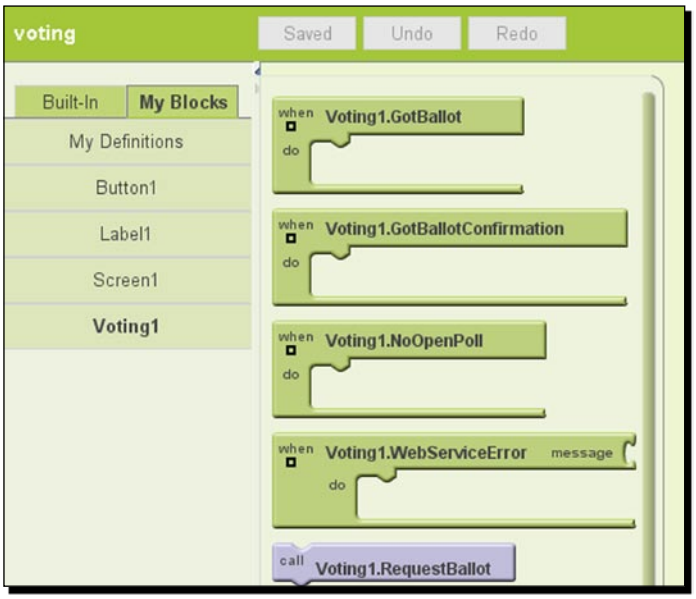
SoundRecorder

As you might guess, this component lets us record sound. We can use the **Player** component in the Media drawer to play it back. We can save it for quick reuse with **TinyDB** or on the web with **TinyWebDB**. You don't actually save the file to `tinydb` or `tinywebdb`. The file that is created is stored on the `sdcard`, and you can then reference the filename to save a link to the file in **TinyDB** or **TinyWebDB**.

We'll do a sound recording app in the next chapter.

Voting

This component is used in apps that track polls and other forms of voting. We'll build a voting app in the next chapter, too.



What we learned

We first looked at advanced components in **Design**'s **Other Stuff** drawer. These included **ActivityStarter** to interface with other applications and more, learning how to use the **BarcodeScanner** component for everything from QR codes to cans of tuna, and setting up Bluetooth wireless communication clients and servers.

We displayed timely messages on the phone's screen using **Notifier** and recognized speech with the **SpeechRecognizer** component. Then, we let our phone read out loud using **TextToSpeech**.

Not Ready For Prime Time is a drawer of things not quite finished but still useful.

FusionTablesControl let us use Google's online Fusion Tables service, and we were introduced to creating games with the **GameClient** component and recording sound with the **SoundRecorder** component.

TinyWebDB is all about interfacing with database services on the web, and we did that and finally met **Voting** as a component for running surveys and so forth.

This completes the basics of using App Inventor. It's time for five chapters jammed with examples of fully-fledged applications.

First, let's build some apps that communicate!

5
Apps That Communicate

Android phones and other devices are all about communicating with the world. This chapter covers apps that call, text, e-mail, and even tweet. In general, we'll build practical apps that communicate in some form or the other, using the basics learned in previous chapters.

In this chapter, we'll learn:

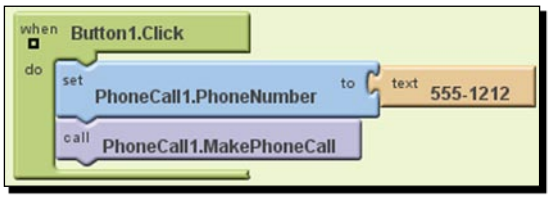
- Building apps that look up phone numbers and make calls for us, and do other neat things related to phone calls
- How to use texting and keep in touch with our friends
- Constructing apps that send e-mails
- Apps for social communication

Good, we're getting into it. Now that we can drag together blocks that do *things* (quite literally the way in which we program using Google App Inventor), let's explore communications capability in our apps.

App—phone home

Tablet computers may be growing in popularity, but the majority of Android devices right now are phones. Phones make calls. So, let's look first at doing a phone app.

Making a call with App Inventor is incredibly easy—we need only a button and the **PhoneCall** component from the **Social** drawer in the **Palette** column of **Design**. As shown in the next screenshot, set it up in the **Blocks Editor**, tap the button on your phone's screen, and it makes the call.



When you disconnect after finishing the call, you'll return to the App Inventor app that initiated the call.

Now, assuming you've done some of the basic tutorials on the App Inventor site (`http://appinventor.googlelabs.com/learn/tutorials`), you've seen the *PicCall* tutorial that uses the **PhoneCall** and **PhoneNumberPicker** components to let us select numbers, see contact photos if they are there, and make calls.

Looks good, huh? The problem is it does not work on a lot of phones yet. Here's the posted warning message:

> *Warning: PicCall does not work on all Android phones in the current implementation of App Inventor: you'll get an error notice on some phones when you try to pick a phone number. Also, you won't see all your contacts — only those created from Gmail. These limitations will be removed in the future.*

I'm sure this will be fixed sometime soon, but, in the meantime, it would be bad practice for us to put out an app that does not work on lots of phones, such as my own Droid 2. So, in our example for using the phone function of our devices, we will use a more reliable method of storing numbers — TinyDB, AI's built-in database making storage of data from session to session easy.

The title of my version of this example app is *CallFriends*.

Rather than doing a lot of tedious formatting with horizontal and vertical arrangements, we'll use a saved AI shell project to speed things up.

Download from Wow! eBook <www.wowebook.com>

Saved projects (saved AI barebone or shell projects) are preformatted App Inventor files shared in zip format, loaded into **Design**, and saved as the name of your new app. Thus, they save you a lot of formatting work. Create and/or collect a directory of these. It makes building apps faster.

The template we're using for our larger example apps in this chapter (**Design** view below) is my *Six Screen Template*. Get it at `http://arrsoft.com/examples/six_screen_template.zip`. Use it freely.

5:09 PM
Six Screen Template by Ralph Roberts
This is my six-page-with-buttons-at-the-bottom template. It's free to use and modify. Enjoy and I hope it saves you labor.\n \nEach of the buttons below change color to show which screen is active. There is a
©2011 ArrSoft... may be used and modified freely!
One
Two
Three
Four
Five
Six

Of course, it has six screens and provides buttons to switch screens. These buttons "light up" by changing their image to show which screen is active. If your app requires less than six screens, the extra ones can be easily deleted, or more screens can be added.

Import ZIP source files into **My Projects** with the **More Actions** / **Upload Source** buttons.

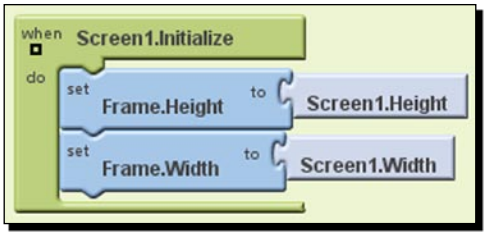
You'll notice in the previous figure that things look a little jumbled. Time for me to remind you that the Design screen's virtual phone does not show WYSIWYG, or *wizzywig* as it's called, the old "what you see is what you get" some programs do. App Inventor, for technical reasons, does not. So, using a test device is important, letting us see the actual look and operation of our apps.

Below is how the *Six Screen Template* looks on a phone. I've included some instructions and description in it as a full-page label on the first screen. After you save it to your app's name, you can delete it. Use this template over and over. I'll also be posting others.

1:10 PM
Six Screen Template by Ralph Roberts
This is my six-page-with-buttons-at-the-bottom template. It's free to use and modify. Enjoy and I hope it saves you labor.
Each of the buttons below change color to show which screen is active. There is a copyright label toward the bottom of the first page. Place your information here. Remove this label and you can change the background color of the screen to whatever you like. See Chapter 5 of my forthcoming book from Packt, GOOGLE APP INVENTOR BY EXAMPLE for ways of using this template.
A 10-pixel margin for top, left, and right is included, and this page area is a vertical arrangement. Just add visible components and use horizontal arrangements for separation. Delete the screens you do not need or add more if called for (see the blocks for how).
Best! --Ralph
©2011 ArrSoft... may be used and modified freely!
One
Two
Three
Four
Five
Six

Here's a trick I use in this app that will prove useful in your toolbox. In the **Screen1.Initialize** block (that is called when the app first starts), set the height and width as shown in the following screenshot. Frame is the name of the vertical arrangement set up to be the container for the entry page or virtual screen of your app (that is, I renamed the virtual arrangement).

This covers you on the many different screen sizes, and your app will look right on lots of devices. **Fill Parent** works also and can be set in **Design** when you first design the app.



Grab that template now, and we'll have it ready for use later.

But, for this first example, *CallFriends*, I've done a lot more for you. I've made available the source for this complete app, heavily annotated with what every block group does. I'll show you where to get it in a moment.

CallFriends app

Our project is an app to call our friends. If your phone is like mine, your contact list is cluttered by all sorts of numbers, both business and personal. And, while you can keep frequently-called numbers, they can be cleaned out if you (as you should) clear history occasionally to free up memory. So, let's build an app for keeping track of just our best friends.

Below is the finished version of my take on this project (the source of which you can download at the link below). You are free to modify it with what we're learning in this book, change the name, and publish it on Market or elsewhere. This one, despite the copyright, is free to use, modify, and distribute. ArrSoft (a Ralph Roberts software) is my own company.

4:10 PM
CallFriends
Call
Add
Edit & Del
Ver. 1.0, ©2011 ArrSoft, arrsoft.com
CallFriends
Reach out and say Hi!
Select a friend and call
You can't stay in your corner of the forest waiting for others to come to you. You have to go to them sometimes
--Winnie the Pooh
More ArrSoft apps
Exit

By the way, most of my published apps are free. You can always find them by searching the Android Market for *ArrSoft*.

As you can see in the previous screenshot, I've fancied my app up a bit. There are lots of ways to do that. Use Photoshop or its free equivalent, Gimp. Download free clipart from the Internet. Or just very neatly use all text.

Let's build this baby. It's more involved than anything we've done before. So, you'll find it much easier if you have the app source.

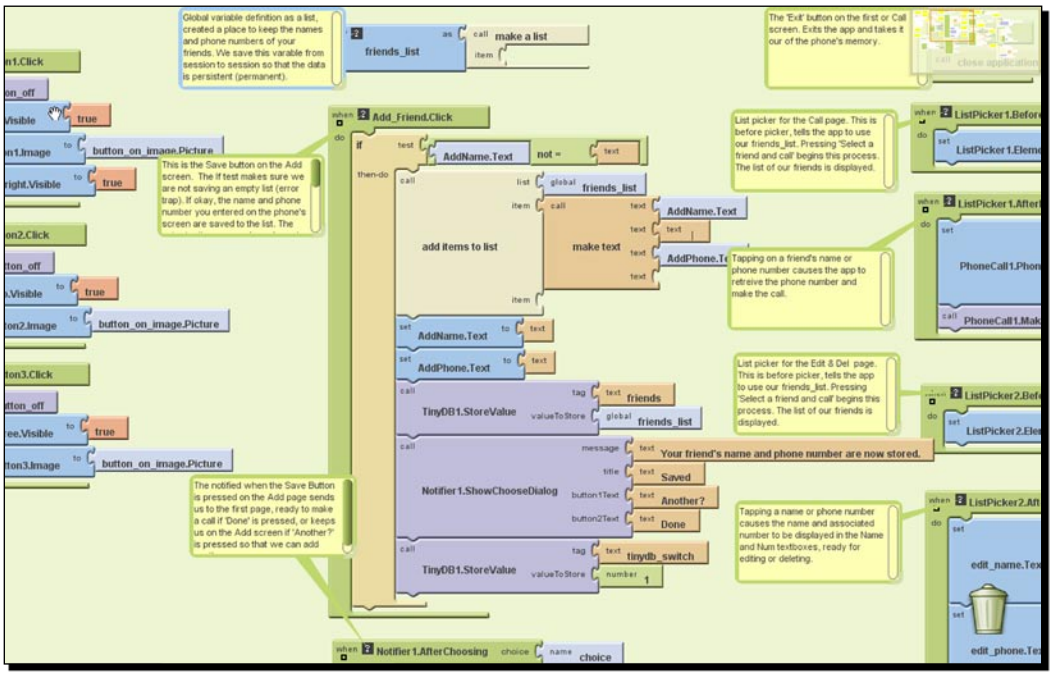
Time for action – getting the source

1. Download the source (a ZIP file) from `http://arrsoft.com/examples/CallFriends.zip`.
2. Upload the zip file to **My Projects** using the **More Actions** / **Upload Source** buttons.

What just happened?

App Inventor's web software portion automatically opened the ZIP file and loaded it into the **Design** screen. You have all the graphics of my completed app and the design of all three screens.

Open the **Blocks Editor** and you'll find (a portion shown in the next screenshot) the complete blocks with lots of explanations. If you don't see the explanation balloons, just left-click on the small black boxes with white question marks. Click on them again to hide the balloon.


Global variable definition as a list, created a place to keep the names and phone numbers of your friends. We save this varable from session to session so that the data is persistent (permanent).
friends_list
make a list
The 'Exit' button on the first or Call screen. Exits the app and takes it our of the phone's memory.
Add_Friend.Click
AddName.Text
not =
This is the Save button on the Add screen. The if test makes sure we are not saving an empty list (error trap). If okay, the name and phone number you entered on the phone's screen are saved to the list. The
add items to list
make text
List picker for the Call page. This is before picker, tells the app to use our friends_list. Pressing 'Select a friend and call' begins this process. The list of our friends is displayed.
Tapping on a friend's name or phone number causes the app to retreive the phone number and make the call.
AddPhone.Text
TinyDB1.StoreValue
tag
friends
valueToStore
List picker for the Edit & Del page. This is before picker, tells the app to use our friends_list. Pressing 'Select a friend and call' begins this process. The list of our friends is displayed.
Your friend's name and phone number are now stored.
Notifier1.ShowChooseDialog
message
title
Saved
button1Text
Another?
button2Text
Done
The notified when the Save Button is pressed on the Add page sends us to the first page, ready to make a call if 'Done' is pressed, or keeps us on the Add screen if 'Another?' is pressed so that we can add
Tapping a name or phone number causes the name and associated number to be displayed in the Name and Num textboxes, ready for editing or deleting.
tinydb_switch
number
Notifier1.AfterChoosing
choice
name

Referring back to the figure before last, the app's first screen, let's consider the design parameters for the *CallFriends* app. What do we need it to do? Well, three basics:

- Select a friend's name and number, make the call
- Add the names and numbers of new friends
- Edit or Delete names

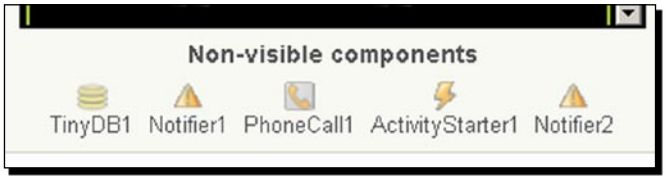
This is an easy way to start creating any app. Decide what it does and write a simple outline.

Based on our outline, we need three screens. Yes, we could have scrunched it all into one or combined the add and edit functions onto one page, but it looks better as three screens—easier for the end user (and always keep their comfort in mined) to understand and intuitive to use.

I started with my six-screen template and removed three screens (just delete the buttons for them and that deletes the blocks also — be *sure* before doing this, it cannot be undone). The buttons were moved to the top because I thought some scrolling might be needed. It wasn't needed, but the buttons looked good up there and seemed to work well, so I left them. Look and feel is more artistic than technical.

Now, it's time to decide which components are needed. And, I am often still surprised by how few we need to build sophisticated and powerful apps. In *CallFriends*, we only need five non-visible components, plus, of course, all the visual design components. Here's what they do:

- TinyDB1: our database to store names and numbers.
- Notifier1: a non-visible component that can show various kinds of alerts and can log information. This is used in the **Add** phase to let the user know a new number was saved and to give them the choice of adding more numbers or returning to the **Call** screen.
- PhoneCall1: makes the call.
- ActivityStarter1: used as my ad button to open Market to my other apps. For easy installation, use this technique in your apps.
- Notifier2: second instance of a **Notifier** component, used in the **Edit & Delete** phase to make sure the user really wants to save (and lets them know that action was successful) or that they really want to delete. This is just good standard programming practice—always provide a way to back out of critical operations as we all make mistakes.



Other components we need on the first screen (the **Call** screen) include the navigation buttons at the top (already there in this case because we used a template), a label for the app's name (change it to one of your own), a way to get the number and call with, and whatever you want down at the bottom. I like including an **Exit** button to politely remove the app from memory and my promotion button for ArrSoft's growing stable of apps.

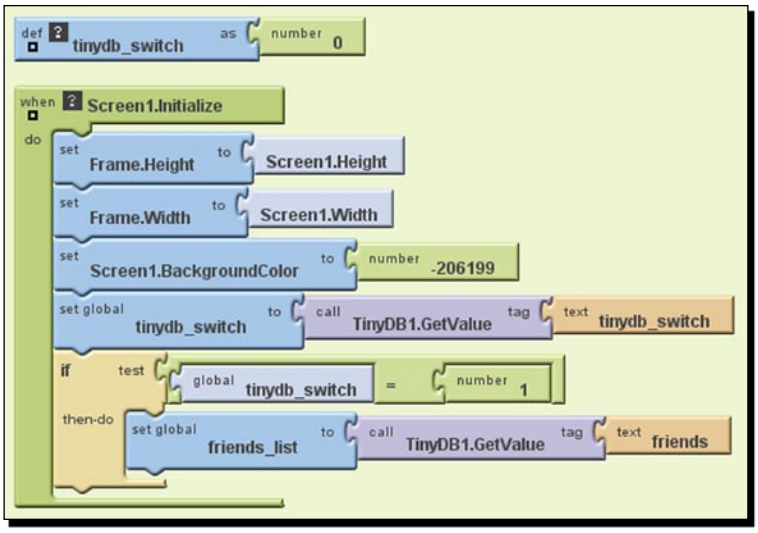
About the **Select a friend and call** button — it's not really a **Button** component. Instead, we use the **ListPicker** component (**Basic** drawer). While the app runs, we have our names and numbers in a list. We save that list to TinyDB for storage when the app is not running and retrieve it when the app gets called again.

Now, we can start laying blocks. Go to the **Blocks Editor**.

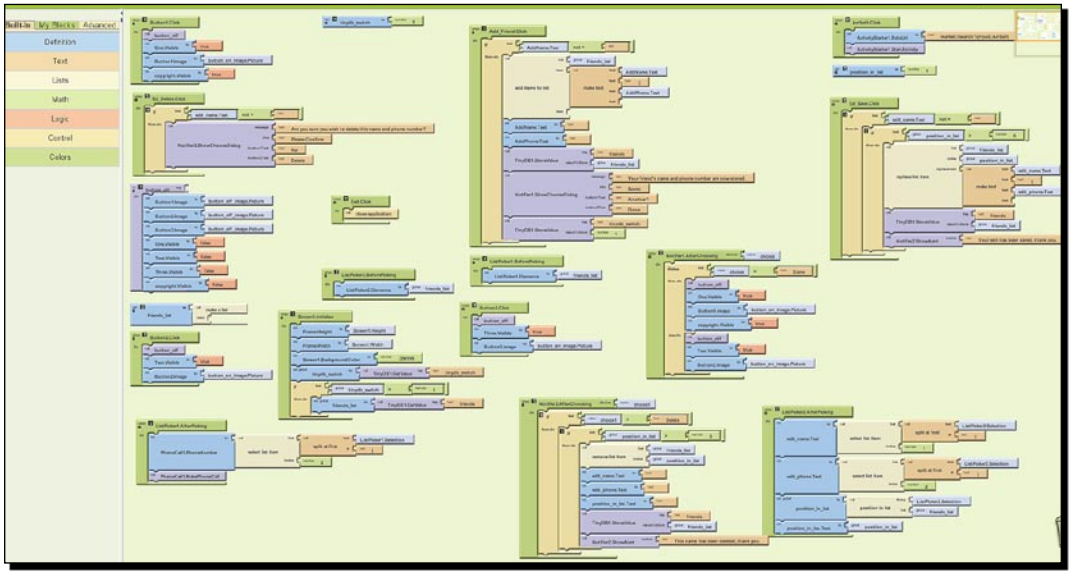
Stuff that happens when the app starts from scratch is set in the **Screen1.Initialize** block (shown in the next screenshot). In our case, the device's display is sized to actual screen size, background color for the entire app is set to -206199 (found with my *Random Colors* app, free on Market) — just a nice-looking color, pleasing to me. The value for **tinydb_switch** is retrieved from the TinyDB component. If the value is `1`, meaning the phone number list has something in it, names and numbers in **friends_list** are retrieved, otherwise its value is left empty until some names and numbers are entered.

The reason for the **tinydb_switch** (a global variable) value and the **if test then-do** control block (shown in the next figure) is an error trap. The necessity for error traps usually becomes apparent when we test the app and we come back later to add the blocks, as I did here.

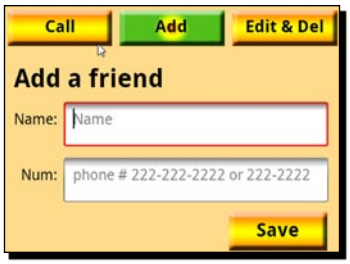
In this case, if the list is empty (like when the app is first installed), an empty string is returned from TinyDB and it causes a crash, killing the app the first time we try to use the **friends_list**. Not good. So, if this switch is `0`, the list is not retrieved. If a name and phone number are added, the switch is set to `1` during the save process and, at the same time, written to TinyDB. If all the names and phone numbers are deleted, the switch is returned to `0`.


def tinydb_switch as number 0
when Screen1.Initialize
do
set Frame.Height to Screen1.Height
set Frame.Width to Screen1.Width
set Screen1.BackgroundColor to number -206199
set global tinydb_switch to call TinyDB1.GetValue tag text tinydb_switch
if test global tinydb_switch = number 1
then-do
set global friends_list to call TinyDB1.GetValue tag text friends

Zooming out on the source blocks for *CallFriends* (which I hope you've loaded into your Blocks Editor by now), we find (see the figure that follows) it divided into roughly four columns. Doesn't matter where you put blocks groups; I just did it this way for ease of understanding.



The first column contains mostly general stuff such as the button groups (from my *Six-Screen Template*), the **Exit** button, and my promo button, as well as the initialization screen—all touched on earlier. The second column mostly concerns what happens on the **Add** screen.



On the **Add** screen (as shown in the previous screenshot), we type in the name and phone number of a friend and tap the **Save** button. This adds our friend to the **friends_list** global variable (defined as a list at the top of the second column, but it could live anywhere on the **Blocks Editor** screen — that is, blocks groups are not linear in positioning).

The **Add_Friend.click** blocks group and the **Notifier1.After Choosing** are what happens when we click the **Save** button. Click on the small question mark on each block group for a description of the steps that each accomplishes. It's a straightforward way of saving data to a TinyDB with error trapping. A technique you can use in many, many apps.

In the third column (block groups concerning the **Edit & Del** screen, shown in the next screenshot), we choose a name and phone number from those already added to edit or delete by clicking the **Choose** button. We make our changes and save with the **Save** button, or remove a name and phone number by clicking the **Delete** button.

The fourth column has groups of blocks that handle the **Delete** and **Save** functions.

Looking at apps one block group at a time and seeing how they work to achieve the overall goals of the app makes learning App Inventor faster and easier. Someone should write a book showing how to learn by examples like this, eh?

So, now you have a complete app, all broken down with lots of notes in it. What now?

Well, I gave you this app source for two reasons—first, because it's a breeze to learn app creation this way, by having real working source; and second, so that you'll have a headstart in modifying these source examples and publishing your own apps. Both fun and a confidence-building learning experience.

Again, this source is yours to modify and use freely.

Have a go hero – modify this app and make it your own

1. Change the title and look of this app in **Design**.

When deciding on a title for your app, first search Android Market to make sure no one else is using the name you want.

2. There is no check to see if phone number is blank or not; add one.
3. If user has not entered name, an alert is needed.
4. There is no check to see if name already exist in **friends_list**. As a result, you may store duplicate names.
5. Also, you may want to consider setting the phone number to **NumbersOnly** as this will keep other characters than numbers from being entered.
6. In **Blocks Editor**, modify the operation by adding extra features or refining existing ones (certainly room for improvement in it).
7. Publish it to Android Market.

App Inventor does not currently allow publishing directly to the Market, but there are two excellent small programs (both free, but donations accepted) that let you publish your app. They are:

- AppToMarket by M. Hossein Amerkashi: `http://amerkashi.wordpress.com/` and `https://groups.google.com/forum/#!forum/apptomarket` for questions and answers
- *Marketizer* by Gene Kupfer: `http://www.taiic.com/marketizer/`

Try both, and decide which works best for you. Sooner or later, App Inventor will include a feature to publish apps. Until then, Hossein and Gene are heroes to a lot of us who wanted our apps widely available.

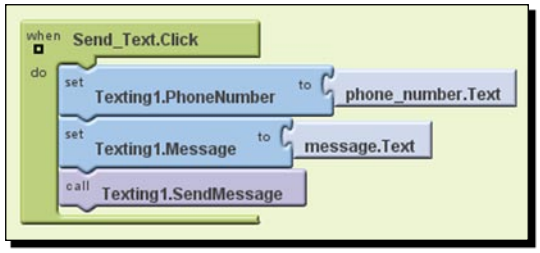
When you finish your version of the *CallFriends* app, we'll move on to texting. No hurry, I need to get another cup of coffee anyway (I can get hundreds of words out of a good cup of coffee!).

Texting with words and more

Texting, as it is popularly called, is really the SMS (Short Messaging Service), a protocol allowing text messages of up to 160 characters to be sent to other phones. Most phones these days at least offer texting as a feature, although not everyone activates it.

The convenience and appeal of texting is its capacity, which lets us send short messages to people without the full interruption of a phone call. It's quite handy.

Sending *texts* (another widely-used term) from App Inventor, using its **Texting** component (in the **Social** drawer on the Design page) is easy. A couple of textboxes to collect the phone number and the message to send, put it in a button like below, tap the button, and AI uses the SMS server on your phone to text.



Of course, easy does not always mean efficient. Yes, AI does texting, but it's more convenient and efficient just to use the texting app that came with your phone for a minimal application like above.

However, unlike the texting app, you can customize and elaborate texting in an App Inventor app to your heart's content.

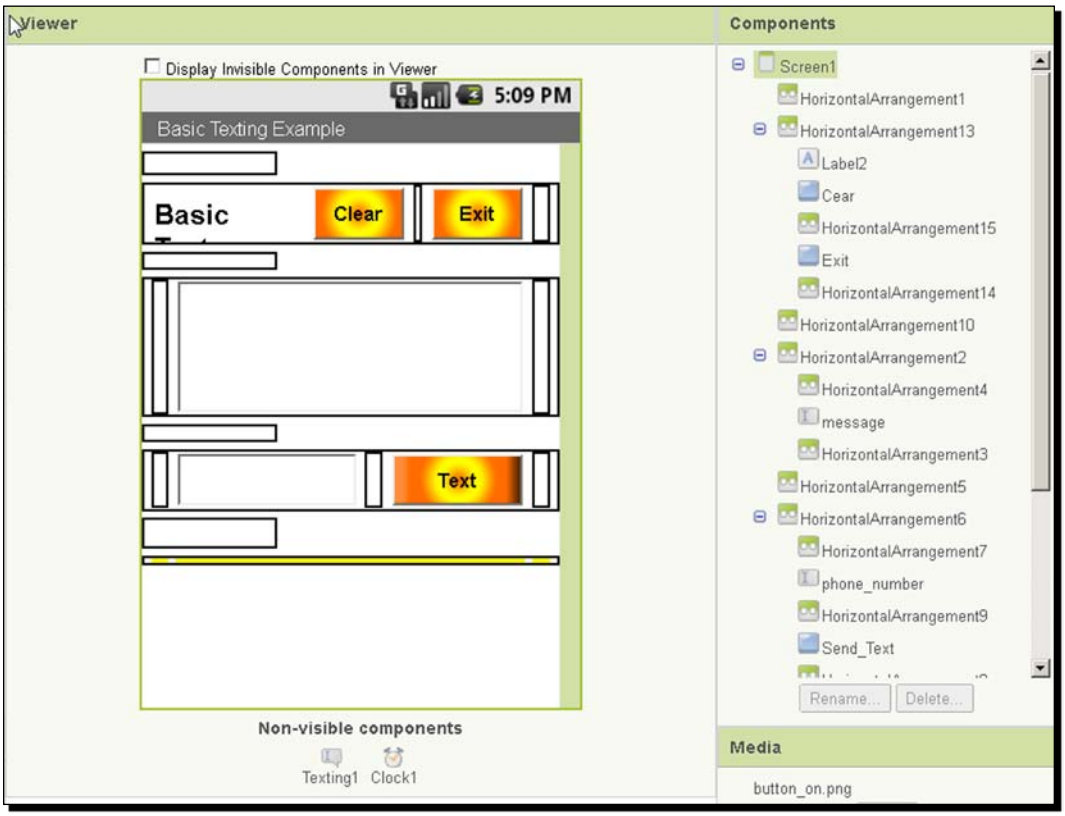
Let's build a texting app, one that you'll be able to build on for your own texting needs.

Time for action – building a texting app

The complete source for this example is at `http://arrsoft.com/examples/texting.zip`.

First, we set up the app in **Design**. For operational components (as opposed to ones that just make the app look better) we need:

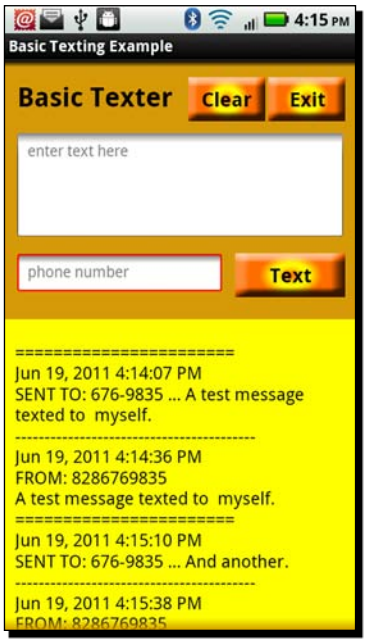
- A button (rename it **Clear**) for clearing out old messages.
- Another button, name it **Exit**. This shuts down the app gracefully by politely removing it from the user's phone's memory.
- A textbox for the message to be sent (I named mine **message**).
- A textbox for the phone number (name it, oh, **phone_number**).
- A button to send the text, called... wait for it ... **Text**.
- Now, a label that I call **receiving**, to accumulate both sent messages and the ones we receive back.
- Finally, in the non-visible components, we need **Texting** and a **Clock** component for timestamping.



In addition to these operational components, I use a lot of horizontal arrangements (as shown in the previous screenshot). They let me space and position elements for a nice, professional-looking app.

You'll develop your own preferences, but I find myself using horizontal arrangements the most. They make the difference between a sloppy amateurish-appearing app and one that makes you look good—and all this costing little additional effort to achieve.

The next screenshot shows what my version of this app looks like. The only outside item I needed was the button graphic (same one for all three buttons). You can use your own or the one here, included in the source file.



Once you have the look of your app to your satisfaction, move to the **Blocks Editor**.

Referring to the source blocks for this app (next), we need to accomplish the following tasks by building groups of blocks:

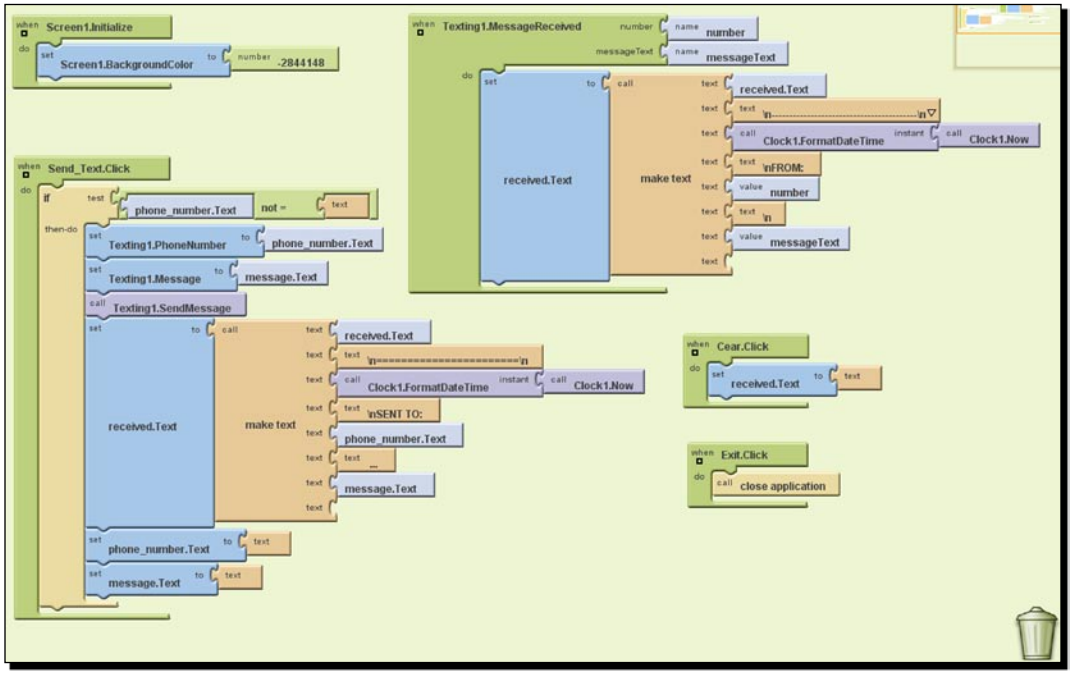
1. In the **Screen1.Initialize** block, set the background color for the app. This can be one of the standard colors from the **Built-In** / **Colors** drawer in **Blocks Editor**, or a custom one such as I used from my *Random Color Picker* app.
2. Now, go to the send text button (the one named Send_Text on the phone's screen). In the next screenshot or my source file, note everything in the button is nested in an **if then-do** block framework. This is an error trap because trying to send a message without a phone number entered causes the app to crash. Anticipate errors and trap them out or else our users will think whoever built the app is an idiot. We don't want that, huh? Also, you may want to add a trap to make sure there is something in the textbox to send, instead of sending an empty text (use the same method as discussed before).

3. In the rest of that framework, the phone number and message textboxes are used to send the message. Then, in **received.Text** we accumulate the sent info, format it, and report it back to the phone's screen. Finally, the two textboxes are cleared, ready for a new text entry.

4. When a text message comes in, the **Texting1.MessageReceived** block gives us the number and the message. We format it into **received.Text** and display it in the received label on the phone. This retains prior sends and receives until cleared.

5. Program the **Clear** button to erase message history by setting **received.Text** to an empty string.

6. Finally, get a **close application** block from the very bottom of the **Built-In / Control** drawer and click it into the **Exit.Click** button framework. This, again, gracefully exits the app. In designing and creating apps, always think of the end user. It makes people like your stuff.



What just happened?

We've now designed and built a usable texting application. Is it worth publishing on the Android Market? No.

Our app works well, but it does *not* do anything that the texting app which came with our phones or a fancy texting app like *Handcent* doesn't do better.

And, there's currently still a shortcoming in App Inventor apps—they only work when active, not in the background like those texting, phone, and e-mail apps that come with your phone or other device. We hope the developers will cure this soon, but, as I've already said more than once, it's still early days in the development of App Inventor. Even so, we still have plenty of power to do some very useful apps.

Well, what would be a worthwhile texting app in AI? How about something similar to our *CallFriends* app from the last section? Yeah! It would be cool to have the capacity to quickly text our buddies, families, sweethearts, and so on.

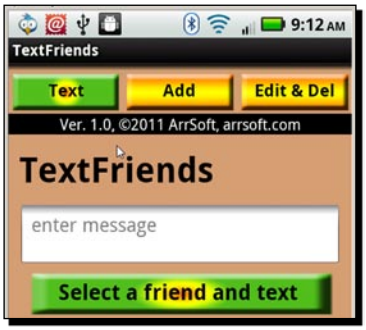
Time for action – create a text-your-friends app

I've already done a lot of the work for you; get the annotated source at: `http://arrsoft.com/examples/TextFriends.zip`.

The major part of both *CallFriends* and *TextFriends* is the **friends_list** variable and the TinyDB database. Both have exactly the same requirements to add, edit, delete, and save names and phone numbers.

So you can just save *CallFriends* as *TextFriends* in **Design**, change the title, change **call** to **text**, add a label to input the message, and you are pretty well done with cosmetics. You don't have to touch the **Add** and **Edit & Delete** screens — they are perfect as they are.

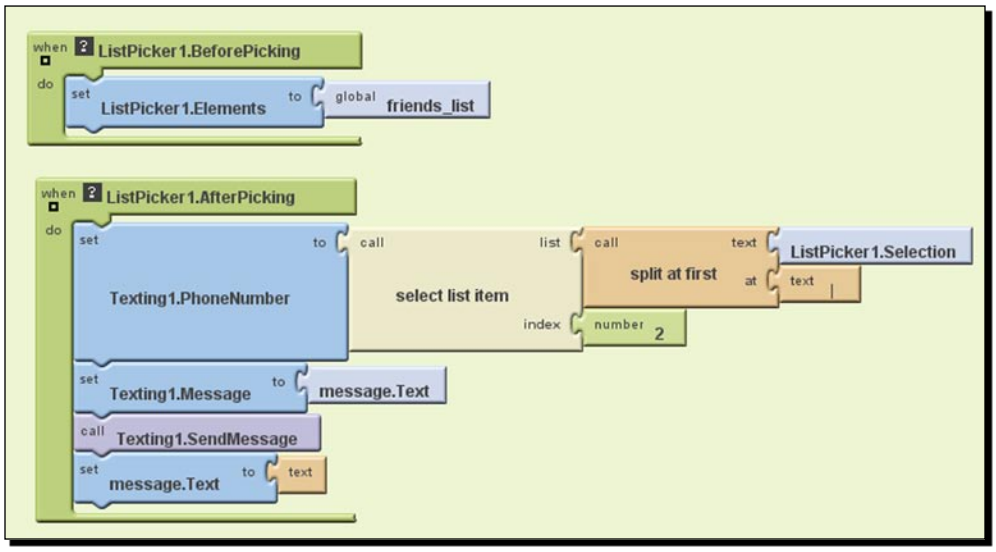
In the non-visible components, delete **PhoneCall** and drop in **Texting**. The finished home screen of the app should now look something like this:



That was simple enough. Now, what if I tell you that modifying the block groups in **Blocks Editor** is just as simple? Yes, it is.

Open (if you have not already) the **Blocks Editor**. Find the **ListPicker1.BeforePicking** and **ListPicker1.AfterPicking** blocks. These are the only things you need to change to repurpose this app from one making phone calls to one that texts.

The phone call stuff is mostly gone already because we deleted it in **Design**. Just change what's there to this:


when ListPicker1.BeforePicking
do
set ListPicker1.Elements to global friends_list
when ListPicker1.AfterPicking
do
set Texting1.PhoneNumber to call select list item list call split at first text ListPicker1.Selection at text index number 2
set Texting1.Message to message.Text
call Texting1.SendMessage
set message.Text to text

What just happened?

We now have a reasonably useful app to text our friends with. Is this one ready for Market?

No. The *CallFriends* app provides two-way communications, *TextFriends* only sends text.

Is it that easy to fix? Well, yeah.

Have a go hero – adding receiving texts

There's not much needed to allow *TextFriends* to receive as well as transmit texts. In fact, it's all in the previous example in this section!

You'll need to add a **Clear** button and a **received** label below the **Select a friend and text** button. You might want to move the **Exit** and promo buttons up above the label so that the sent and received texts can scroll without pushing them down out of sight. Do a trifle of formatting with horizontal labels, and then go to the **Blocks Editor**.

Drag in the **Texting1.MessageReceived** framework (find it in **My Blocks** / **Texting1**).

Format and fill **received.Text** as we did in the Basic Texting Example, and you're good to go.

One of the secrets to prolific programming is accumulating routines you can re-task and use in other apps.

Having fun yet? Well, let's e-mail someone about that!

Applications e-mailing stuff

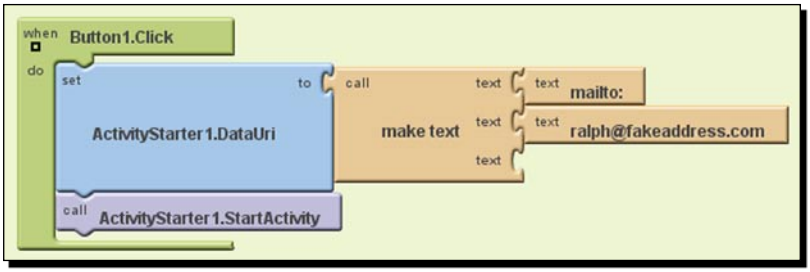
The big three, the *trifecta* of smartphones, communications-wise are phone calls, texting, and e-mailing. As in the two just covered, the apps that come with your phone or tablet will do the basics of these better than an App Inventor app.

But, of course, we have the flexibility, as we did above, to add our own magic touch, and we'll do that here.

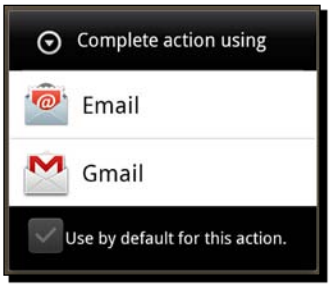
First, here's how simple it is to send e-mail from an App Inventor app. In **Design**, you need a button and drop the **ActivityStarter** in as a non-visual component. In the **Properties** column for it, type the same information as we use to call a web page, `android.action.intent.VIEW`. Leave everything else blank.

In **Blocks Editor**, use the button click frame block for whichever button you want to use for sending the e-mail. With only one in my example, that would be **Button1.Click**, but it could be **Button26.Click** or, if you renamed them as you should with that many buttons, perhaps **Send_Email.Click**.

Fill your button as shown in this screenshot:



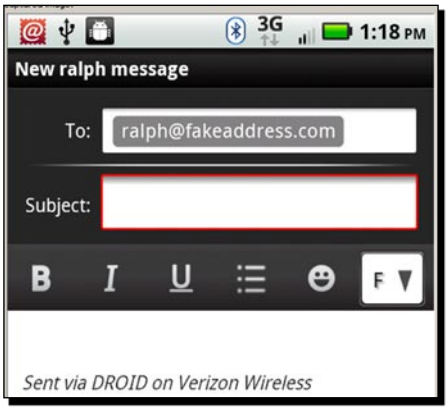
Tapping the button causes App Inventor, via **ActivityStarter**, to call your phone's e-mail client app. In my case, I have both regular e-mail and Gmail, and I can choose either one.



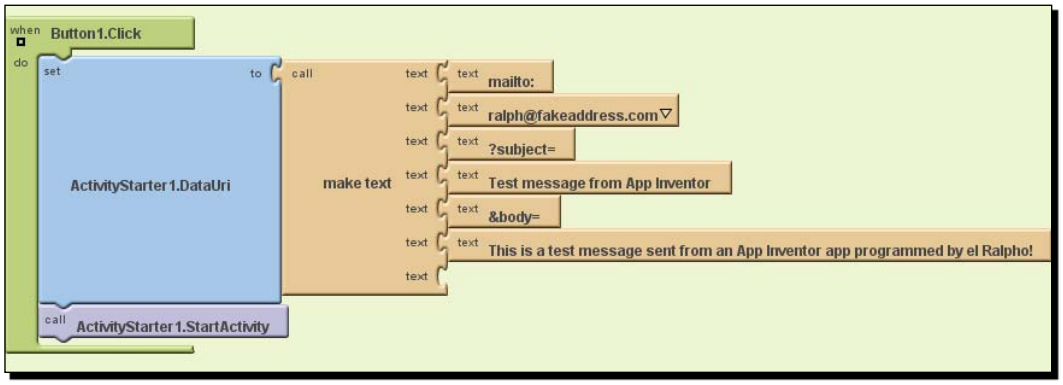
And, I chose regular e-mail because I have my own mail servers and use that the most.

Now, the e-mail client comes up with the address already in place. I type the subject, the body of the message, and send it.

Sending or cancelling the e-mail returns you to the App Inventor app.



We can do more than just call the e-mail client. By using the **make text** block as below, we can construct a standard `mailto:` URL with subject and message body. The mail client is still called and **Send** must be tapped, but the e-mail is all filled out otherwise.



While App Inventor can't completely send e-mail on its own, or at least not yet, we can still control e-mail functions with it.

By the way, the reason I use `ralph@fakeaddress.com` above is that I learned a long time ago as a computer book author, *do not* put your real address in an example. About 17,000 people a day will test the example and your mailbox fills up. However, I have no objection to you knowing my real e-mail, `ralph@abooks.com`, and you are welcome to e-mail me with valid comments or questions. I enjoy getting feedback.

Now, what about receiving e-mail?

No...not yet. You can set up an AI app to open your e-mail client to check mail, then return. If you recall the Activity Starter & Intents app I recommended in *Chapter 4*, *Mastering Concepts and Advanced Components* that's a good starting point. Remember my analogy of the hole in the floor that let Java code peep through. Here, for example is what you need for an ActivityStarter to open Gmail for you:

- Action: `android.intent.action.MAIN`
- ActivityPackage: `com.google.android.gm`
- ActivityClass: `com.google.android.gm.ConversationListActivityGmail`

Which brings us back to the question we asked in the sections on phone calls and texting: given these constraints, what app could we do that would be interesting and useful?

Here's what I came up with: *I_Love_You!* — it's an app that lets you send messages to your sweetie prefaced with love quotes. Let's do it!

Time for action – that Loving feeling

The basic premise of the *I Love You* app is simple enough. The app picks a random endearing quote and sets up an e-mail for us to our beloved.

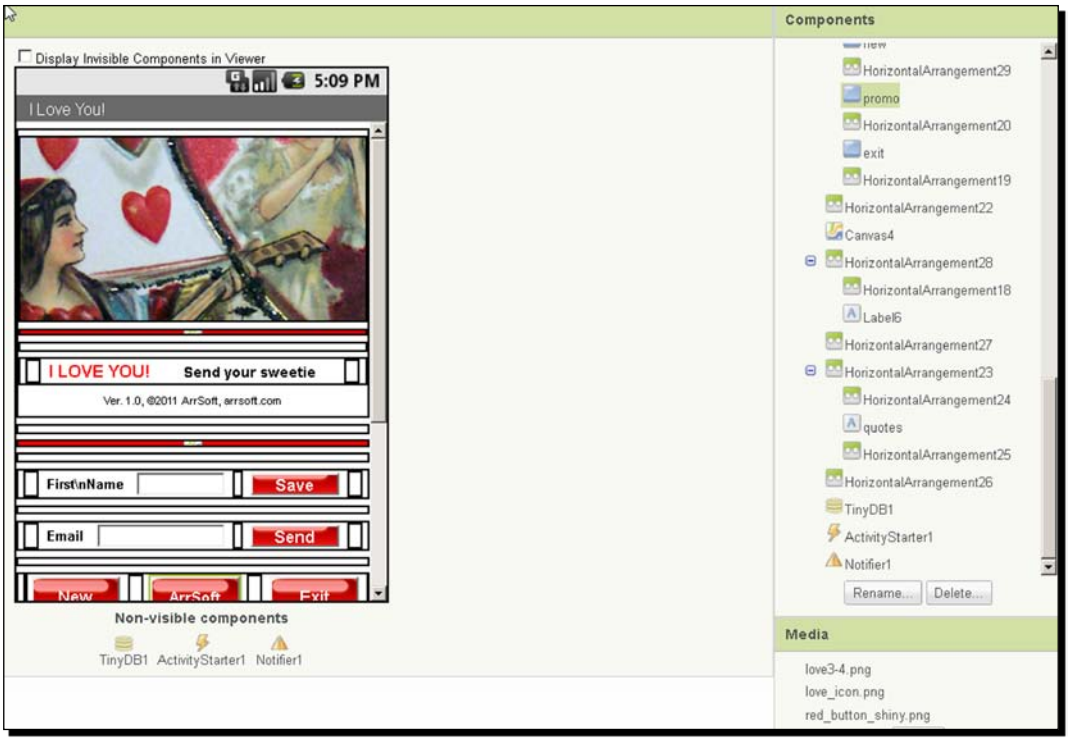
Starting is easy; just find a bunch of public domain quotes about love on the Internet. In fact, you'll find a lot about love on certain websites, but please do not get distracted, eh?

The complete annotated (love that word) source is at: `http://arrsoft.com/examples/I_Love_You.zip`, and you might want to download it now for ease of following along. Then, use it as a blueprint and template for you own version.

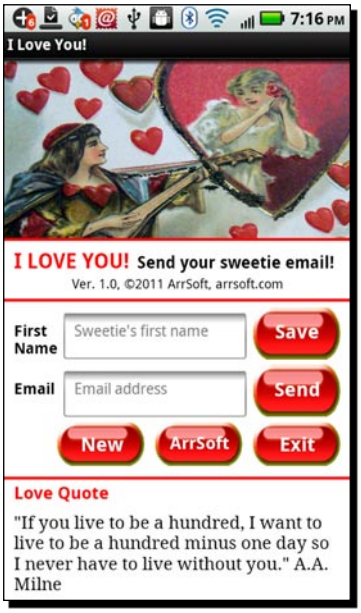
Setup in **Design** is not complicated at all — in fact, most App Inventor apps use a minimum of components for a maximum of accomplishment.

In *I Love You!*, we need a title, two textboxes, five buttons, and a label for the quote. And, of course, we add our usual sprinkling of horizontal arrangements and the like for neat professional-looking formatting. Something similar to this:

Download from Wow! eBook <www.wowebook.com>



So that when it appears on your phone's screen, the finished app is like so:


7:16 PM
I Love You!
I LOVE YOU! Send your sweetie email!
Ver. 1.0, ©2011 ArrSoft, arrsoft.com
First Name
Sweetie's first name
Save
Email
Email address
Send
New
ArrSoft
Exit
Love Quote
"If you live to be a hundred, I want to live to be a hundred minus one day so I never have to live without you." A.A. Milne

What tasks do we need to accomplish? This is a minimal, one-screen app. We want to pull blocks together for the following tasks:

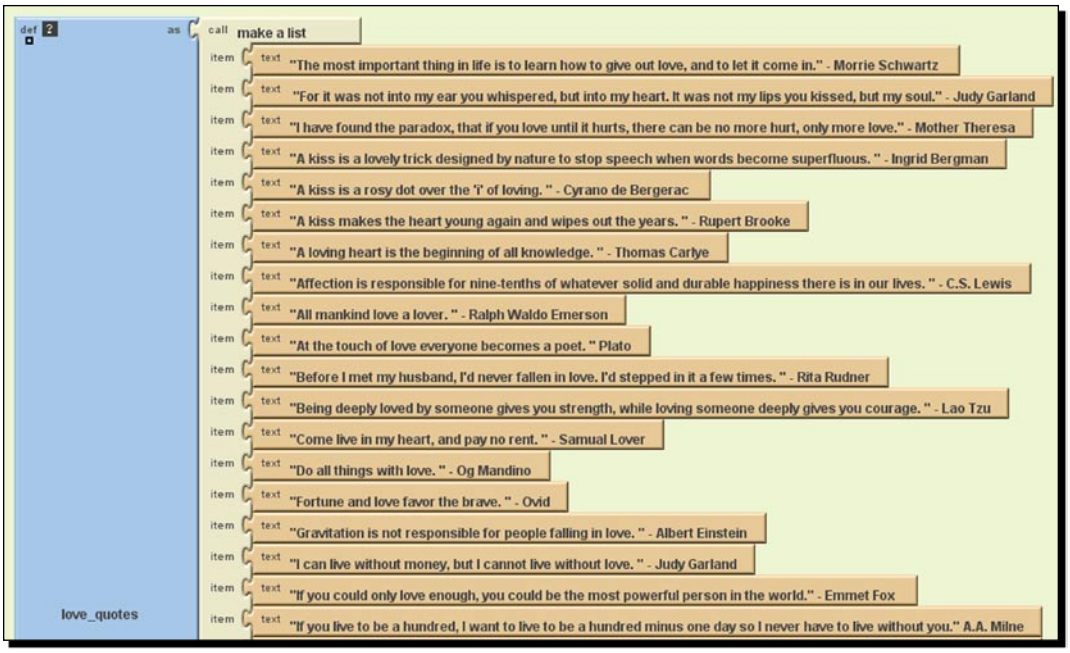
1. Enter the user's sweetie's first name and e-mail address. This is a one-time process (unless the user is playing the field), and we will use TinyDB to make this persistent.
2. Once the **Save** button is pressed, writing the name and e-mail to TinyDB.
3. When the screen first comes up, we have a quote displayed at the bottom of the screen. If the user wants to find a more appropriate one, pressing the **New** button displays another random quote from the library of quotes we'll put into the app.
4. The **Send** button opens the phone's e-mail client and starts an e-mail with sweetiepie's e-mail and the subject line "I Love You!" (which you can change to suit yourself).
5. The **ArrSoft** button is my promo (this app is already on Android Market). Change it to whatever you like or leave it out entirely.
6. And the **Exit** button is our kind way to remove the app from memory when not in use.

We also need a **TinyDB** non-visible component for storage, an **ActivityStarter** for e-mailing, and a **Notifier** for putting up error messages and letting the user know that saving was successful.

And that's the entire design of the app.

For the block groups that enable all this to work, please bring up the source ZIP you downloaded in the **Blocks Editor**.

We first define a global variable, **love_quotes**, as a list (just snapping the **make a list** block from **Built-In** / **Lists** does the defining) to give us a place to put our love quotes into.


def 2
as
call make a list
item text "The most important thing in life is to learn how to give out love, and to let it come in." - Morrie Schwartz
item text "For it was not into my ear you whispered, but into my heart. It was not my lips you kissed, but my soul." - Judy Garland
item text "I have found the paradox, that if you love until it hurts, there can be no more hurt, only more love." - Mother Theresa
item text "A kiss is a lovely trick designed by nature to stop speech when words become superfluous. " - Ingrid Bergman
item text "A kiss is a rosy dot over the 'i' of loving. " - Cyrano de Bergerac
item text "A kiss makes the heart young again and wipes out the years. " - Rupert Brooke
item text "A loving heart is the beginning of all knowledge. " - Thomas Carlye
item text "Affection is responsible for nine-tenths of whatever solid and durable happiness there is in our lives. " - C.S. Lewis
item text "All mankind love a lover. " - Ralph Waldo Emerson
item text "At the touch of love everyone becomes a poet. " Plato
item text "Before I met my husband, I'd never fallen in love. I'd stepped in it a few times. " - Rita Rudner
item text "Being deeply loved by someone gives you strength, while loving someone deeply gives you courage. " - Lao Tzu
item text "Come live in my heart, and pay no rent. " - Samual Lover
item text "Do all things with love. " - Og Mandino
item text "Fortune and love favor the brave. " - Ovid
item text "Gravitation is not responsible for people falling in love. " - Albert Einstein
item text "I can live without money, but I cannot live without love. " - Judy Garland
item text "If you could only love enough, you could be the most powerful person in the world." - Emmet Fox
item text "If you live to be a hundred, I want to live to be a hundred minus one day so I never have to live without you." A.A. Milne
love_quotes

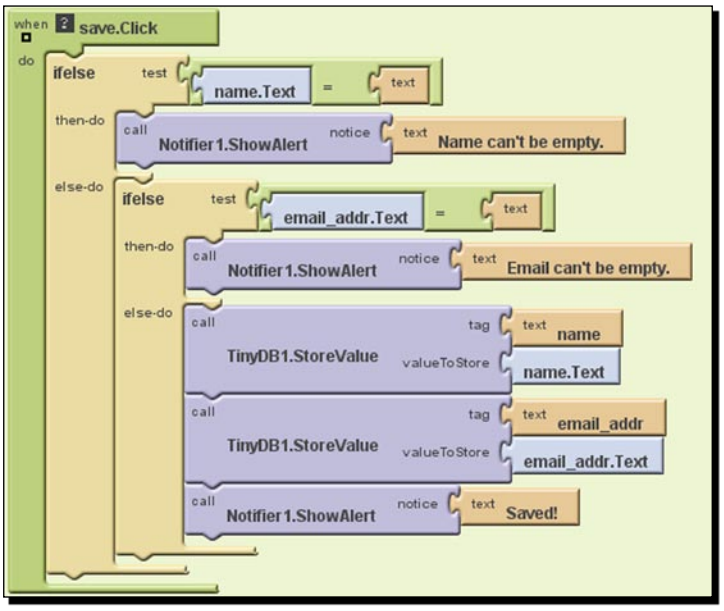
There is a limitation in using lists for data storage in App Inventor; right now, it's only possible for a list to contain 55 items in the **make a list** block. However, you can have several lists and use the **Append to List** block (from **Built-In** / **Lists**) to join lists together when the app first starts (do it in the screen initialization block). For lists with hundreds or thousands of items, store it on the web and use the **Web**, **TinyWebDB**, or **Fusiontables** components to retrieve it (which we'll get to in *Chapter 6*, *Apps That Remember*).

We put two textboxes in **Design** for the sweetie's name and e-mail. When the **Save** button is pressed, we error trap for empty name or e-mail strings, which would crash the app and show an alert as to which one is missing (using the **Notifier** component). This is done by nesting an **ifelse** framework from **Built-In** / **Control** inside another — thus testing for empty strings in both textboxes and letting us notify the user which is empty.

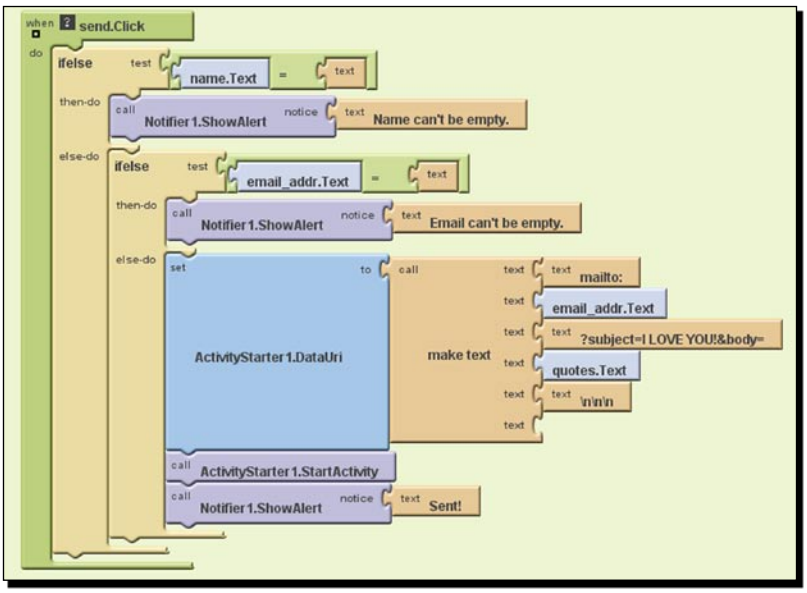
If both textboxes have data (name and e-mail), each value is stored in TinyDB by using a tag (naming the storage field) and recording that along with the textbox value.

Finally, the notifier assures the user it was saved. And, again, the name and e-mail will be remembered (appear in the textboxes) from session to session until changed by the user.

Here's how it looks in the **save.Click** frame (the one for what happens when the **Save** button is tapped):



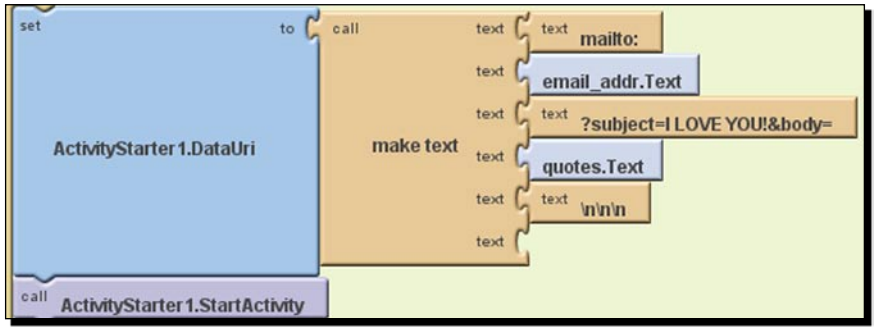
The only other block group of any size is for the **Send** button. We use the same error trapping as in the **Save** button (nested **ifelse** frame blocks). If both values are present, we build the e-mail and sent it to the phone's e-mail client for mailing, as shown here:



Sending e-mail is such a basic thing you'll want your apps to do, a little review won't hurt us here.

We use the ActivityStarter to set up e-mail. In **Design**, in the **Action** field for this component, enter `android.intent.action.VIEW`. Exactly like that.

For the **ActivityStarter1.DataUri** frame block (see below) in **Blocks Editor**, we use a **make text** block to construct a standard `mailto:` URL. This is not App Inventor programming, but just plain old web stuff using formatting and variables that e-mail and other web services understand.



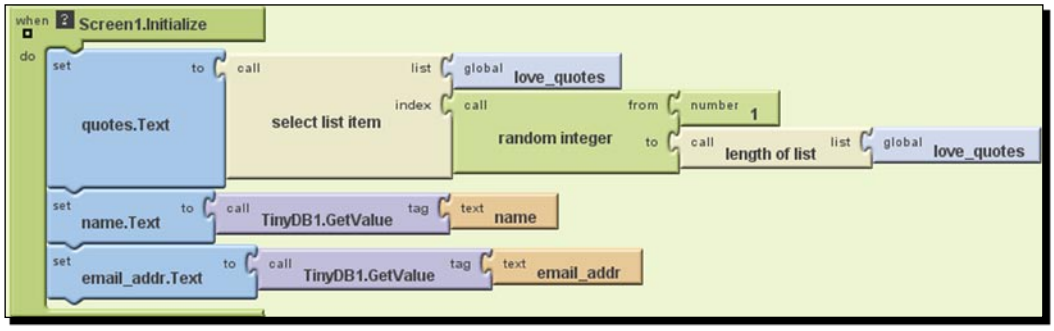
That done, we just have a few little things left.

Now that we know what our apps save, we can construct the screen initialization block.

So, in **Screen1.Initialize**, when the app starts up, we pick a random quote and get the name and e-mail of our sweetie from TinyDB.

In selecting the quote randomly, instead of counting and keeping track of the number of quotes in the list, we select it based on a random integer from 1 to (using **the length of list** block as below) however many quotes there are. Let AI do the work of counting and tracking; we've got better things to do — such as sending sweet nothings to our sweetie.

In this block, the contents for the two on-screen textboxes are retrieved from TinyDB.



Three final small block groups to cover (all depicted below) — the **Exit**, **Promo**, and **New** buttons.

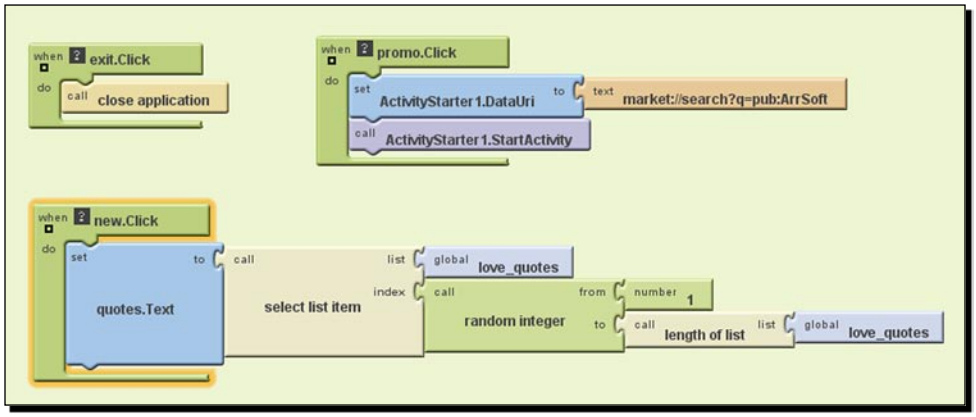
The **exit.Click** frame block defines what happens when the **Exit** key is tapped. This allows the app to exit gracefully and remove itself from memory. I've stressed this point several times that it pays to be nice to the end user. Think of their limited resources.

The **Promo** button we've looked at before. In my case, I open *Market* on the user's phone, showing all my apps ready for installation or giving me a five-star rating, eh?

In this app, we can use the same activity starter as the one calling the e-mail client, so no extra resources get added to the app. Both require only `android.intent.action.VIEW` in the **Action** field (found in the **Properties** column on the **Design** screen).

Last, we have the **New** button, which lets the user choose another random quote. It uses exactly the same blocks as we used in the screen initialization.

In fact, just left-click on **quotes.Text** in the **Screen1.Intialize** block, type *Ctrl* + *C* to copy, click anywhere on the **Blocks Editor** screen, type *Ctrl* + *V* to paste the **quotes.Text** block with all its sub-blocks onto the screen, grab **quotes.Text**, and click it into the **new.Click** block.


when exit.Click
do call close application
when promo.Click
do set ActivityStarter1.DataUri to text market://search?q=pub:ArrSoft
call ActivityStarter1.StartActivity
when new.Click
do set quotes.Text to call select list item list global love_quotes
index call random integer from number 1
to call length of list list global love_quotes

What just happened?

That's it for e-mailing our sweetie.

We've designed a nice screen, pulled together the blocks that save our sweetie's name and address with error traps, built a list of random love quotes, built the e-mail URL that goes to the e-mail client on the user's phone, given them a means to get new quote, and to exit gracefully. A little promo is also thrown in, because if the user likes this app, they will probably also download more of your published apps.

You have to love App Inventor by now! It's a sweetie, too.

Okay, on with communication apps. Time to look at social media (web services that let you interact with friends or communities). App Inventor has a Twitter component, so we'll start with that one.

Social communication

Social communication is communicating with a group instead of an individual. Twitter, Facebook, and others are all web services facilitating group exchanges.

Twitter lives on the simple end of the social media spectrum, devoted to 140-character messages called *tweets*. I honestly thought this kind of silly when I first heard of it; now I use Twitter almost every day — it's great for promoting projects and just letting folks know what you are up to. Follow me at `http://twitter.com/ralphr`.

App Inventor has a **Twitter** component in the **Social** drawer of the **Palette** column on the **Design** screen. Drop this non-visual component on the virtual screen, and a drawer of blocks relating to Twitter operations automatically appears in **My Blocks** in **Blocks Editor**.

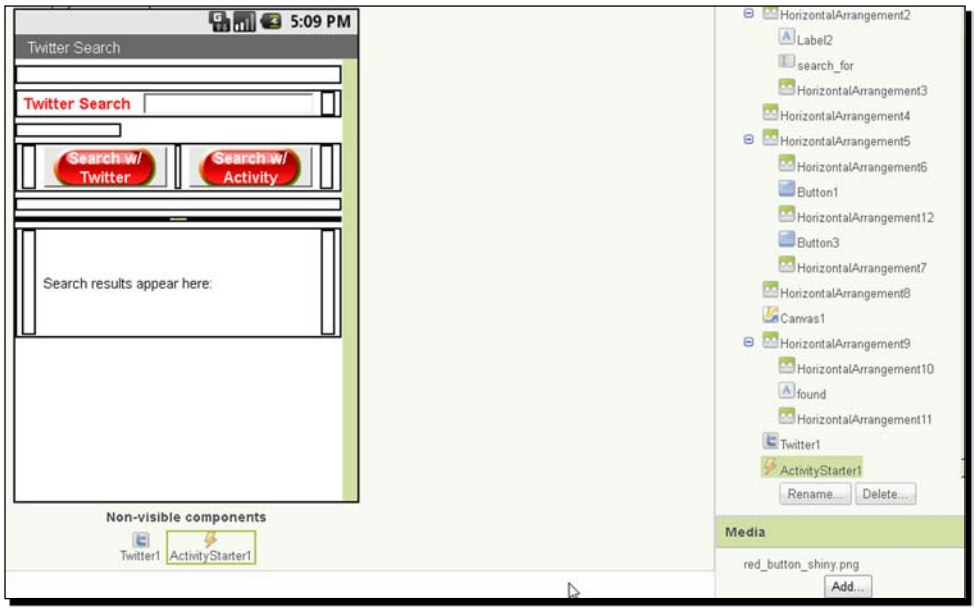
As we keep getting reminded, App Inventor is still very much in development. The **Twitter** component shows potential, but is not as powerful as it will be. That said, it will pay off later for us to be familiar with what it can do now.

The best way to show that, of course, is with an example app. This app, I simply call it *twitter*, shows how to do a search of Twitter using the **Twitter** component. Then, we do the same search faster and easier and with more sophisticated results using **ActivityStarter**.

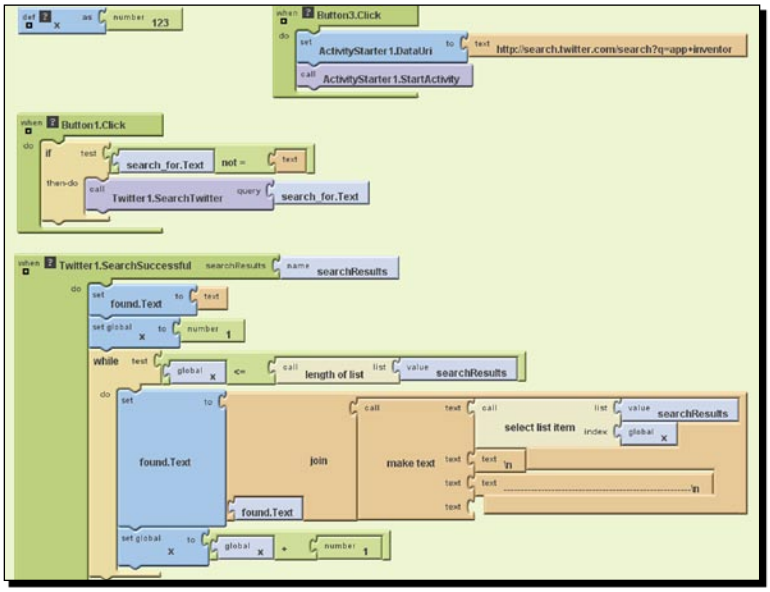
Time for action – two methods for searching Twitter

We start by formatting a nice screen (get the source for this one at `http://arrsoft.com/examples/twitter.zip`). The basic components needed are a textbox for the search term (see the next screenshot), a couple of buttons (for the **Twitter** component search and one for the **ActivityStarter** search), and a label for the search results. Also, drop in the non-visual components for **Twitter** and **ActivityStarter**.

Always take a little time to format the screen with horizontal arrangements—it's great practice.



In the blocks editor, the left column is all required for searching Twitter with the **Twitter** component. **Button3.Click** does the same thing currently better with just those two interior blocks (see the next screenshot).



To build your own or modify mine, here's what the block groups do:

1. Define global variable **x** to use in displaying a list of results from Twitter component search. It serves as an index to point to a specific item in a list.
2. For **Button1.Click**, use the **if then-do** control block as an error trap to avoid an empty string (which causes a non-recoverable error to the app), then use **Twitter** component to search for search term from the textbox on the phone's screen.
3. Use the **Twitter1.SearchSuccessful** framework block to clear **found.Text**, set **x** to `1`, and loop while there is something in the list to read, joining each item to the list and formatting to make readable. The result is displayed on the phone in the **found** label.
4. In **Button3.Click**, we set up the **ActivityStarter** search. This is all needed to do the same search using **ActivityStarter** and viewing it in the phone's web browser. For the purposes of this comparison, we've "hardwired" the search term into the URL.

What just happened?

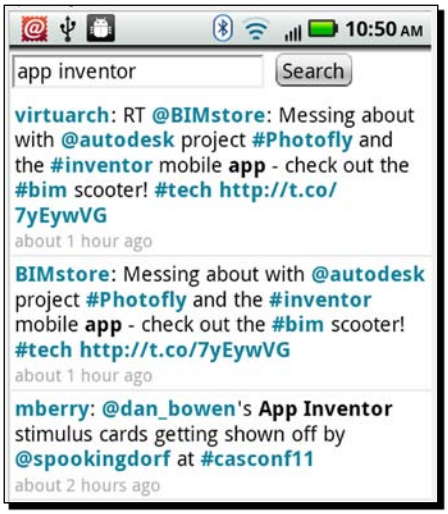
Now, we can use our phone to compare the two.

First, here's the result using the **Twitter** component:

10:48 AM
Twitter Search
Twitter Search
app inventor
Search w/ Twitter
Search w/ Activity
virtuarch RT @BIMstore: Messing about with @autodesk project #Photofly and the #inventor mobile app - check out the #bim scooter! #tech http://t.co/7yEywVG
BIMstore Messing about with @autodesk project #Photofly and the #inventor mobile app - check out the #bim scooter! #tech http://t.co/7yEywVG

Looks okay, but it has some disadvantages, the big one being only the 15 most recently tweeted results returned.

In contrast, searching with **ActivityStarter** (see next screenshot) opens up the web browser on the phone, gives us a more readable format, the links are active (tap on them to follow), additional searches are easy, and you can use the Twitter API (application programming interface) to refine searches, get more and older results, and so on.


10:50 AM
app inventor
Search
virtuarch: RT @BIMstore: Messing about with @autodesk project #Photofly and the #inventor mobile app - check out the #bim scooter! #tech http://t.co/7yEywVG
about 1 hour ago
BIMstore: Messing about with @autodesk project #Photofly and the #inventor mobile app - check out the #bim scooter! #tech http://t.co/7yEywVG
about 1 hour ago
mberry: @dan_bowen's App Inventor stimulus cards getting shown off by @spookingdorf at #casconf11
about 2 hours ago

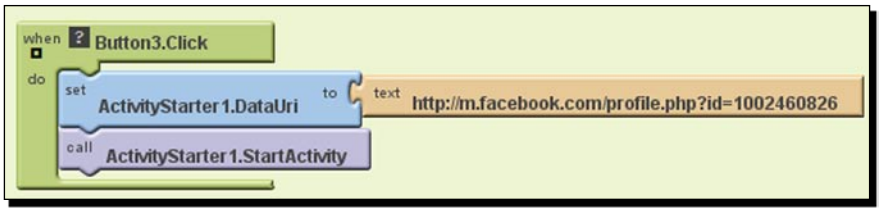
Don't get me wrong, I'm not saying abandon the **Twitter** component. Good things are coming, and even now it has more power than the previous example might lead you to believe.

If you have a Twitter account, for example, you can go to `https://dev.twitter.com/` and register your app, receiving a *consumer key* and *consumer secret*, allowing you to build apps that use Twitter's full power.

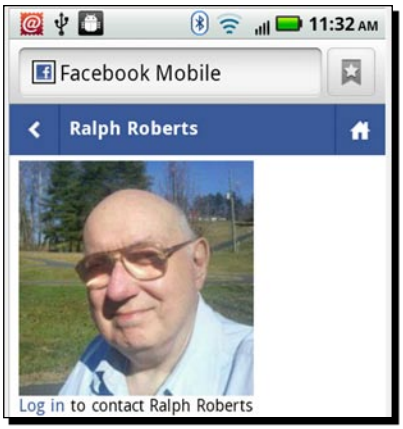
Facebook and other social media sites

App Inventor does not have components yet for Facebook and the other social media sites. However, most of those sites have mobile interfaces that allow us wide scope in using stuff from them in our apps.

Our old friend **ActivityStarter** comes in handy here. Take Facebook. Putting an `m.` in the front part of the URL (see below) causes the result to be formatted for a phone's smaller screen.



So, if I search for my Facebook account, it pops up formatted for my Droid 2.



As popular as these social sites are, I believe we can count on the App Inventor developers to add components for more of them.

A bonus template

Here, just to give you a bonus, is my six-button template with the buttons on top. It's useful of apps where you need the screen to scroll down to show more material. Get it at `http://arrsoft.com/examples/six_screen_template_buttons_top.zip` and enjoy.

Pop quiz

1. The majority of Android devices we will be creating apps for today are:
    a. Tablet computers.
    b. Desktops.
    c. Phones.
    d. Laptops.
2. In order to make phone calls with App Inventor, we need:
    a. An Internet connection.
    b. To use the **phonecall** component.
    c. A special phone account.
    d. All of the above.
3. The Texting component allows us to:
    a. Bug all our friends with lots of texts.
    b. Use the device's Texting to send texts.
    c. Send texts to ourselves.
    d. All of the above.
4. Viewing web pages and sending e-mail:
    a. Should not be done in the same app.
    b. Requires complex programming.
    c. Cannot be done.
    d. Both use the same activity starter action.

Summary

In this chapter, we learned about apps that look up phone numbers, make calls for us, and do other neat things relating to phone calls. And, we saw how to use texting and keep in touch with our friends.

We also constructed apps that send e-mails. And, we searched Twitter in a couple of ways and looked at using other social media sites in our apps.

Now, we move on to the apps that remember; that being the various ways we can access and manipulate data with our apps.

6
Apps That Remember

Your phone, my beloved Droid 2, or any Android device for that matter is really a small (yet powerful) computer. Computers excel in collecting and manipulating data.

In this chapter, we devise apps that remember data (any kind of information) for us, help us collect it, and present it back to us by whatever criteria we specify. The apps we pull together in this chapter (and the following one) will remember stuff (accumulate and return data) both locally and by accessing the Internet.

Here, we will explore examples of:

- Lists and lists of lists and how to use them
- **TinyDB**, persistent to please us, saving data from session to session
- The **Web** component, reaching out to the web

And, in the next chapter (along with other topics), we will cover:

- Fusion Tables, Google's free online data service
- TinyWebDB, a powerful way to use the Internet to access and store data

So, what is it again we were going to discuss? Oh, yeah! How to remember or, more importantly, mastering the ways our apps store and retrieve data.

Download from Wow! eBook <www.wowebook.com>

Lists and Lists of Lists

Lists, of course, we've already met in *Chapter 3*, *Playing with Blocks* but let's look at them in the context of this chapter on the ways App Inventor lets us use data.

Data is any information used by an app. A grocery list is data. The names of states are data. Addresses and phone numbers of your friends and business associates are also data. And any group of related items is a list.

The term data, by the way, is plural. An individual piece of information is a datum, which hardly anyone uses anymore.

More complete lists of data, where each datum is a related list of items pertaining to the same thing, is called a database. Database items are called records. A record has fields (items of the related list). A collection of records is a table.

Here's an example of a database: a contact list of your friends and associates as mentioned above. Each person would be a record. Each record would contain related fields such as name, address, city or town, state or province, country, zip or postal code, phone number, e-mail address, and may include a comment field so you could add information such as "dentist" or "plumber."

Now, we come full circle back to the topic of this section: lists and lists of lists.

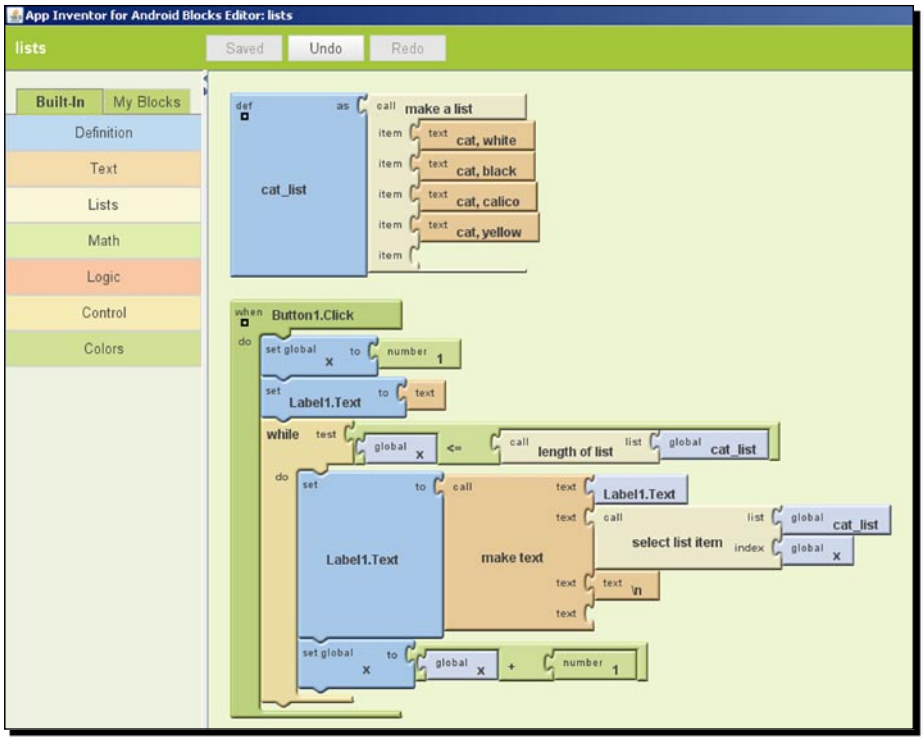
Lists are the simplest form of databases. Lists of lists are lists with lists nested within. The latter correspond to database records and fields—more on that in a moment.

Lists in App Inventor are used in three ways—static lists, changeable lists, and as database record handlers.

Static lists

First, lists are a way to store small sets of static data, such as the list of types of cats used as an example in *Chapter 3*, *Playing with Blocks*, where we first explored how lists work. This method works well when you need to have access to data that is used like constants in programming (always the same)—that is, we won't be changing the list, just referring to it.

The following figure shows a simplified technique of storing and retrieving from a static list:



The blocks above displays the complete list on our phones as follows:

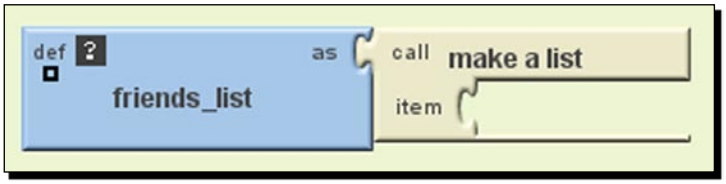
Because this type of list is static, we define it as a global variable (**cat_list**) and it is always in the app. Using the methods described in *Chapter 3*, *Playing with Blocks*, we can search for and pull out items in the list, or the entire list as in the preceding example. Again, the list is always the same, so there is no need to save it.

This is useful, as stated above, for small sets of data and is "hardwired" into the app (i.e., always constant). But, what if we want users of the app to have the capacity of adding and/or removing items from the list?

Changeable lists

The second way of using lists is changeable lists. We let the user modify the list on the fly.

A good example of a changeable list is the way we set up the example apps CallFriends and TextFriends in *Chapter 5*, *Apps That Communicate*. In both of those apps, we simply defined **friends_list** as a global variable, specifying that it is a list variable by plugging in the **make a list** block from **Built-In/Lists** (as shown in the following screenshot):



That's all we do in creating the place for the list. We never enter any data directly into the list as is done in a static list (by plugging in text blocks as those with the types of cats above).

Instead, we set up blocks that allow the user to enter data for him or herself. Of course, we specify what that data is and do whatever error trapping is necessary to keep the app from crashing. Let's define a simple contact list of our friends.

Time for action – building the input screen for a Friends List

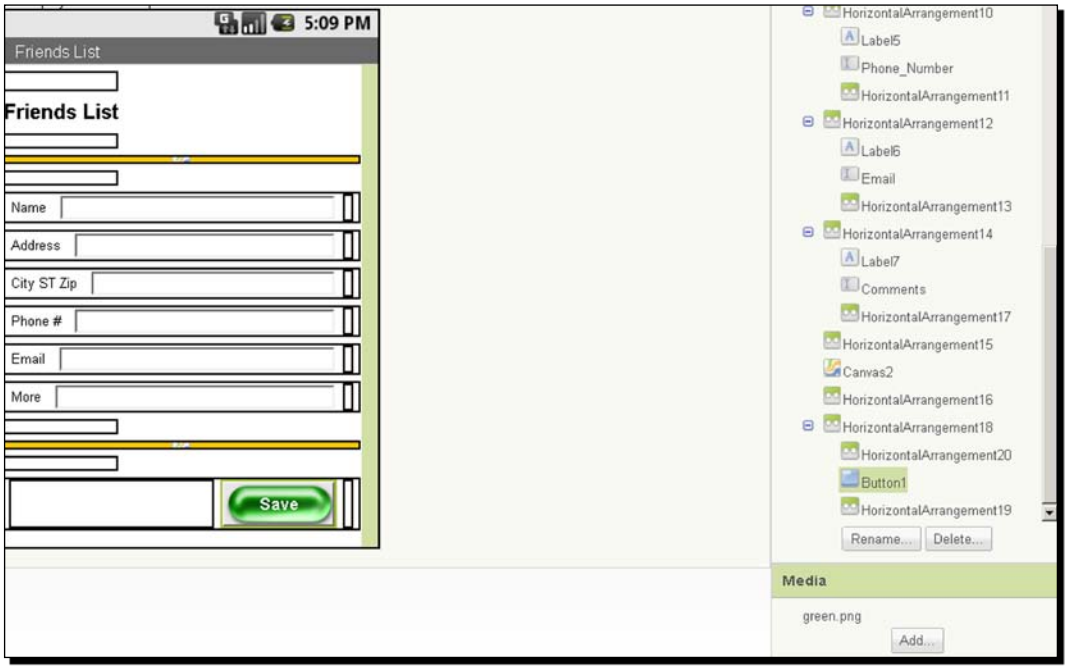
First, we make it pretty, then we make it work.

Create a new project on the **My Projects** page of App Inventor's online component, **Design**. When the **Design** page opens, build the page we'll use to enter the names, addresses, phone numbers, and e-mails of our friends. For the non-visible elements, drop in a **Notifier** component (to let us know the record was saved okay).

This part, of course, is only one screen of what's needed for a full app, but this is the "Add a Friend" screen.

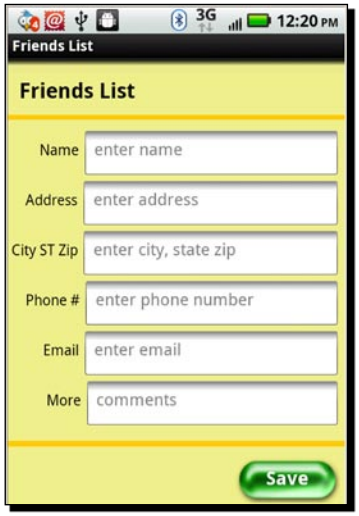
We need a label for the title, six labels and six textboxes for the fields (remember that one name and its associated data is a record), and a **Save** button. We drop those components in and use horizontal arrangements to space things out and make our app look professional (not just a good habit but a necessary one in becoming the fantastic app creator you can easily be using App Inventor).

So, we wind up with something like this. I like fancy buttons, so I uploaded an image and used it for the button.



Bonus: Taking a little extra effort to use fancy buttons makes your app look really good. Here are some free ones I made for you (that's where I got the preceding green button used): `http://arrsoft.com/examples/colored_button_set_1.zip` (unzip these into a folder on your computer and upload single button images as needed).

On your phone's screen, you should have something like the following:

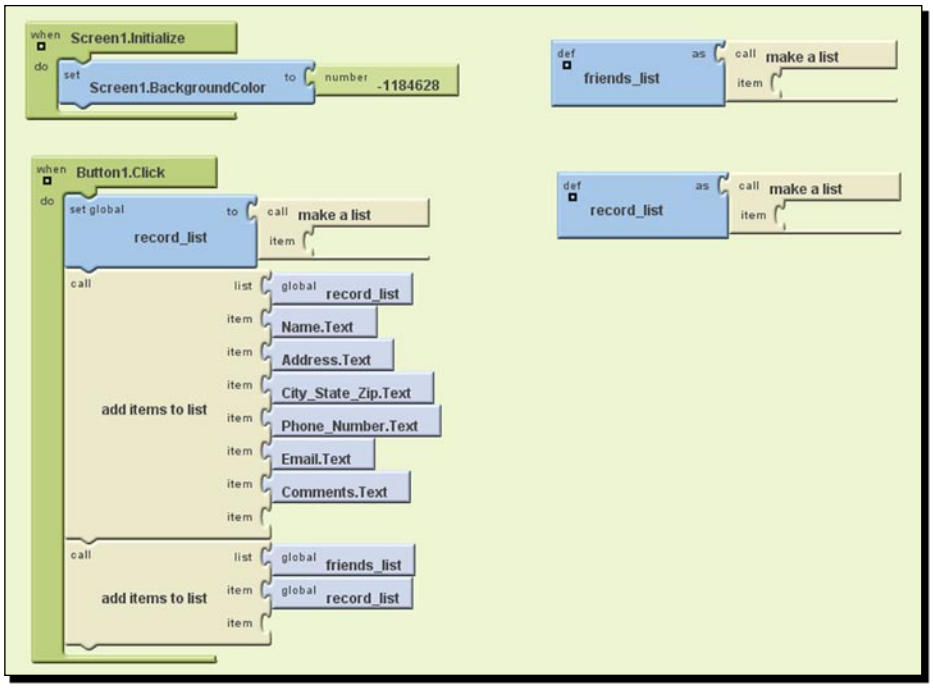


Now, we'll go to the **Blocks Editor** and make the pretty part do actual work by dragging the right blocks into the right places.

In the following figure, we set the screen color (just to be fancy, using my random color selection app; search for ArrSoft on Market, get it for free).

You'll need two global lists variables: one for the entire list (**friends_list**), and **record_list** as a temporary holder for each record as it is entered.

In the button, we clear the temp record list to make it ready for use, then build the record from the six input textboxes on the phone screen—**Name**, **Address**, **City ST Zip**, **Phone #**, **Email**, and **More**. The complete record is then added to **friends_list**, completing the operation.


when Screen1.Initialize
do set Screen1.BackgroundColor to number -1184628
when Button1.Click
do set global record_list to call make a list item
call add items to list list global record_list
item Name.Text
item Address.Text
item City_State_Zip.Text
item Phone_Number.Text
item Email.Text
item Comments.Text
item
call add items to list list global friends_list
item global record_list
item
def friends_list as call make a list item
def record_list as call make a list item

The previous groups of blocks work fine, allowing entry and saving our six-field record. However, it is not production-ready. Production-ready means someone not familiar with App Inventor or our various and unique app creation styles can use it without problems.

Thankfully, once you have the basic operation worked out (like we do in the previous screenshot), adding the extras bringing the app up to "production-ready" is a breeze. And, we all need to be in the habit of thinking this way—that is, "how can the user screw it up?".

In our record entry, this type of list storage accepts empty strings (I found this out as you should, by testing for it). So, no error traps needed here as there were in the examples in the previous chapter.

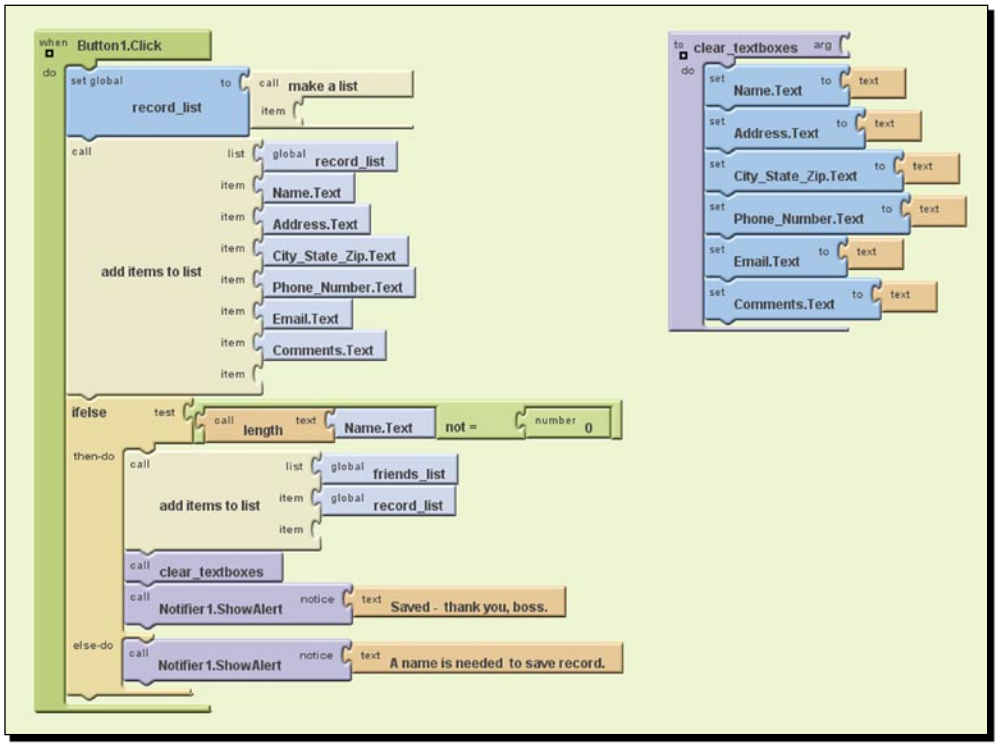
However, I do see two problems:

1. The user could enter useless records by not specifying the friend's name.
2. Notifications of the above, and showing the user that the save process was successful, are needed (otherwise the dude will just keep hitting Save and get a bunch of redundant records).

So, (see the following image) we can use an **ifelse** control block to test for a zero-length **Name** textbox and a **Notifier1.ShowAlert** to let the user know he or she needs to enter a name.

If the record is okay to save (has a name entered), the notifier says **Saved – thank you, boss.** (hey, no reason your apps should not be polite), and the **clear_textboxes** procedure is called to clear all the textboxes and return them to showing the gray hints.

Now, this screen is production-ready (unless *you* think of something else).



What just happened?

We've put together a nice data entry screen to add our friends, their addresses, phone numbers, e-mails, etc, and made it production-ready. This is only part of a full application. Here are the tasks for turning it into a complete app. And, yes, it's similar to CallFriends and TextFriends from *Chapter 5*, *Apps That Communicate*, but we're doing more with lists, fields, and records.

Have a go hero – completing the Friends List application

1. Move everything we have so far into a vertical arrangement, and name it **Add_Screen**.
2. Add two more vertical arrangements named **Home_Screen** and **Edit_Delete_Screen**.

3. On the home screen, use listpicker to retrieve the record for a friend, and pull out the six fields in variables.
4. With buttons, set up choices to e-mail (if there is an e-mail address), text (if there is a phone number), or call (need a number for that, too). From CallFriends, you already have the basis for achieving this.
5. On the edit and delete screen, pull the blocks together so that records can be displayed (in textboxes), changed, and/or deleted.

To help you out, I've set up all three screens for you and completed the TinyDB persistence save and retrieval (I'll show you that part shortly). Download the ZIP with annotated blocks at `http://arrsoft.com/examples/friends_list.zip`, and upload it to your **My Projects**. Here is what it looks like:

Handling database records

Just there, I snuck in what we're now discussing: using lists to handle database records. It's always good to think of data in terms of records and fields. Then, we're ready to write apps that provide information from within the app (lists and lists of lists) or downloaded from the Internet using the **Web** component, or from Google's Fusion Tables, or from a variety of sources using **TinyWebDB**.

Let's see how we get data out of downloaded database tables (remember, a table is a collection of records, and records have fields of data). This applies also to data from spreadsheets.

Perhaps the most common way databases and spreadsheets export data is in Comma-Seperated Values (CSV). CSV are sometimes called comma-delimited files because commas are the most widely used character to separate files.

If we download a table of addresses in CSV format from an online database, it might look like this:

```
Frank Funkywunkerburger,34 Bucky Street,Frigid,AK,11111

Marvin Muckledoodle,1457 Dirty Blvd,Dohickie,IA,22222

Sallie Knockout,977 Beautiful Lane,Goosefat,GA,33333
```

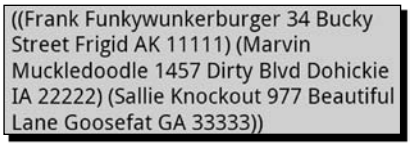
Fields in the records are separated by commas. End of lines separate records.

To use data in this format, you have to know or find out the database structure. In this example, there are five fields to a record—Name, Address, City, State, Zip. Now, we can use it, but first let's get it into a format App Inventor understands.

You guessed it—lists! To be even more precise, a list of lists where the items in a list are themselves lists, giving us an equivalent of that ubiquitous (really, really common) database structure—table, records, fields.

Our comma-delimited CSV file is downloaded from the online database into a variable (in the example here, **addresses_from_a_database**). In the **Built-In/Lists** drawer of the Blocks Editor, we find a **list from csv table** block. We use it as shown in the following screenshot. Basically, just plug it into a label, and plug the variable with the CSV table into it.

Next is a display of how this data—now converted from CSV to a list of lists—looks:

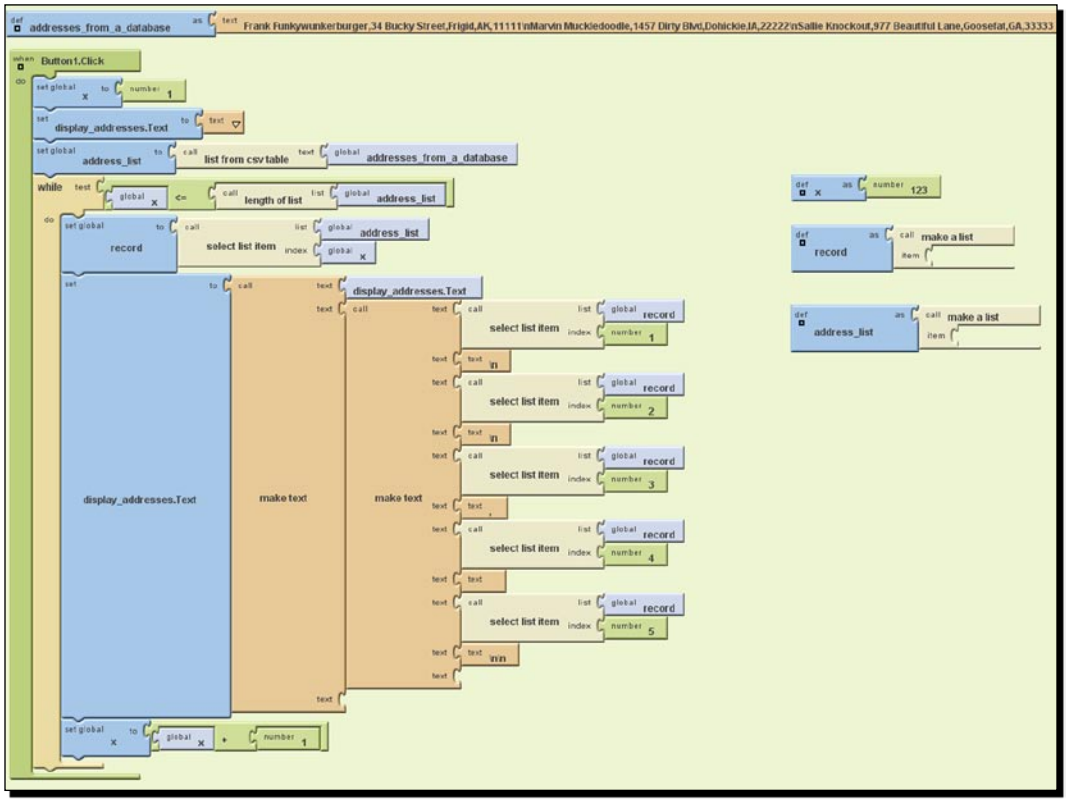

((Frank Funkywunkerburger 34 Bucky Street Frigid AK 11111) (Marvin Muckledoodle 1457 Dirty Blvd Dohickie IA 22222) (Sallie Knockout 977 Beautiful Lane Goosefat GA 33333))

As we see, we now need to break it out of the preceding lists within a list format in order to use it.

In my following example (get it at `http://arrsoft.com/examples/database.zip`), we use a button to activate the conversion/breakout process. Let's do it together.

Time for action – converting a CSV table

In **Design**, we need only a label to display our result and the button to start it. We'll build the blocks:


addresses_from_a_database
Frank Funkywunkerburger,34 Bucky Street,Frigid,AK,11111\nMarvin Muckledoodle,1457 Dirty Blvd,Dohickie,IA,22222\nSallie Knockout,977 Beautiful Lane,Goosefat,GA,33333
when Button1.Click
set global x to number 1
set display_addresses.Text to
set global address_list to call list from csv table text global addresses_from_a_database
while test global x <= call length of list list global address_list
set global record to call select list item list global address_list index global x
set display_addresses.Text to call make text text display_addresses.Text
call make text text call select list item list global record index number 1
text in
call select list item list global record index number 2
text in
call select list item list global record index number 3
text ,
call select list item list global record index number 4
text
call select list item list global record index number 5
text \n
set global x to global x + number 1
def x as number 123
def record as call make a list item
def address_list as call make a list item

Now, to the **Blocks Editor**—the preceding may look complicated, but it's not. It's a nice method to know and which you can adapt for getting usable data from or sending information to any kind of database.

We now need the preceding blocks or block groups to convert our data into an address list:

1. A global variable, **addresses_from_a_database**, mimicking a CSV table downloaded from the Internet (you can use the example in the `database.zip` file referenced previously).
2. Also, define two global list variables—**address_list** and **record**. The first of these holds the converted table, and the second is a temporary holder for each record as it is read in (name, address, city, state, zip).

3. When the **Get Addresses** button is tapped, we set up blocks for these functions:

a. **x** is set to `1` (to begin the loop), **display_addresses** is cleared for use (otherwise it retains stuff from previous operations).

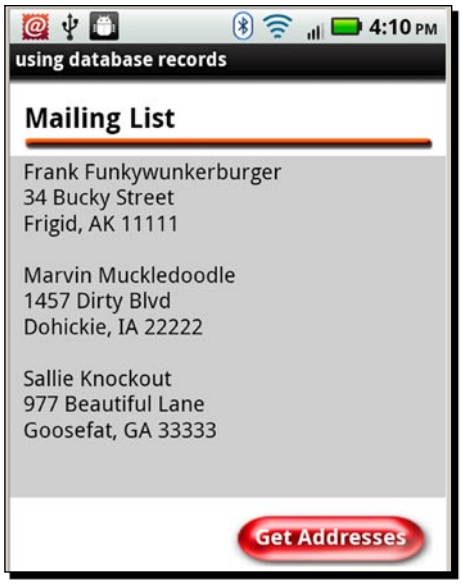
b. Next, the downloaded CSV table is converted using the **list from csv table** block into a list of lists in **address_list** where there is a list (equivalent to the CSV table) of lists (equivalent to the record), and each of these record lists have items equivalent to fields for name, address, city, state, zip. The `\n` is used as a separator for each record.

c. Now, we put in a **while** loop (see the blocks plugged into the test socket), which runs for however many items are in the converted address list because it is using 'length of list' block to determine number of records OR number of times the loop should be executed.

d. Each record is added to the display label used to make text and formatted with line return (or comma and space in city, state, zip), and followed by two line returns to show a blank line between records.

e. Finally, our counter **x** is incremented by `1` to get the next record.

The result looks like this:



What just happened?

We now know how to convert CSV tables, records, and fields to a list of lists ready to use in our apps. This technique will serve you in good stead in communication with online databases and using the data from them with the **Web** component, Google's Fusion Tables, and the **TinyWebDB** component—all three of which are coming up in this chapter.

First, however, once we get data into a global list variable, such as **address_list** in the example above and **friends_list** in the previous to that example, we need to make it persistent (available from session to session). So, we review that next.

TinyDB—persistent to please

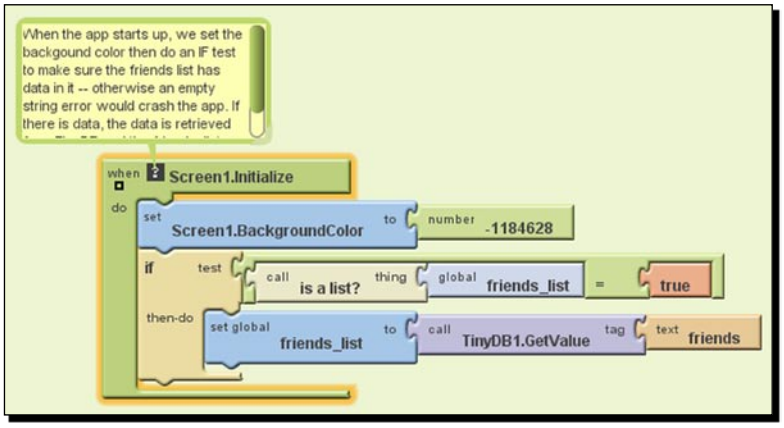
In several examples already in this book, we've seen how to use **TinyDB** (non-visible visual component in the **Basic** drawer on the **Design** page) to make values permanent (that is, persistent).

TinyDB is very simple, requiring only a tag (a name) and the value. After we exit from an app and start it again, the value is still there in **TinyDB** and can be retrieved using the tag. Think of the tag as a filename and the data as a file. Loading the data back into our app is like loading a file—call it up by name, and it comes down off the web and into whichever variable we designate.

As promised in the previous Friends List example, here's how you save the entire list. Pull out a **TinyDB.StoreValue** block and click it into a button, a procedure, or any other frame block. Click in a name and a value and both get saved. As I said before, very simple, as we see here:

To retrieve the value, just plug **TinyDB1.GetValue** into the global variable (as shown in the following screenshot). In Friends List, we error trap to make sure it is a list, but the actual retrieval is just one block and a tag.

Persistence does not work in the emulator, so we need to test that on an actual device such as a phone.



Now, let's reach out onto the world weird web and get some data. Three methods exist in App Inventor for accomplishing database access. We'll look at the newest and my favorite, the **Web** component, first.

Web component—reaching out to the web

The most recent component in App Inventor—as this is written—is the **Web** component. Not to be confused with a browser, this component allows sending or pulling things from the web.

For example, we can grab a complete web page or, rather, the source for the page as App Inventor again is not a browser, although it does now have a webview component, which we'll cover later. So, what you get is a mess of HTML code like this:

Download from Wow! eBook <www.wowebook.com>

However, that can be quite useful, and we're going to find out how in mere moments.

But, first, the **Web** component can manipulate web data in three ways:

1. **Parsing**: we can use AI's built-in text and list blocks to parse (break into parts) that mess of HTML (like the preceding) and extract useful information.
2. **Get**: Get is a method of sending and receiving information to databases, or at least languages such as PHP that talk to databases on web servers for you. Ever notice when you type in a search term on a website, the URL changes to something like `http://arrsoft.com/webtest.php?one=test&two=anothe%20test` ...? The one and two are variables, and the data following the equal signs are the contents of those variables.
3. **Post**: Post is a more secure way to send and receive data from the web; the variables and their contents do not show in the URL, and hence it is more secure. Unfortunately, the post method is not quite ready for prime time in **Web** yet (our heroic AI developers are working on it). However, the first two work quite well.

So, let's play with parsing and using the Get method for sending and receiving data—but parsing first.

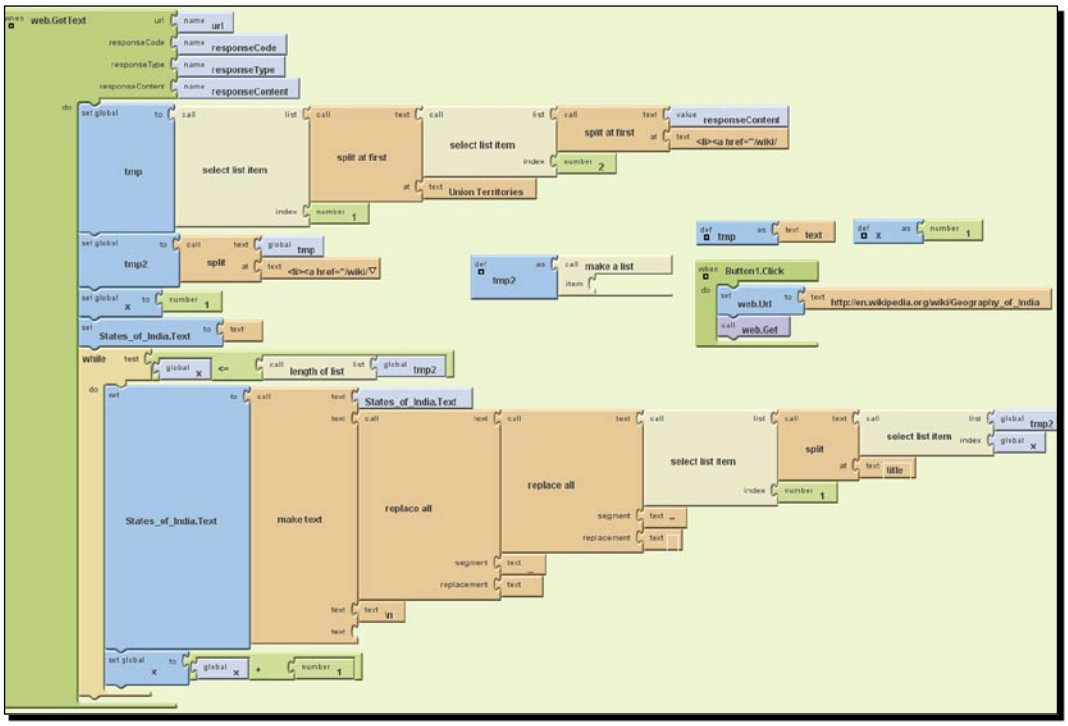
Parsing

Both the HTML source code of a web page and the method used to break out information in a readable format appear complex at first glance. They're really not. Just simple pattern recognition, and App Inventor provides us with the tools we need.

To begin, look at this Wikipedia page: `http://en.wikipedia.org/wiki/Geography_of_India`.

I have some wonderful friends in India, and it will be fun to pull out a list of the States of India from this interesting web page about the geography of India.

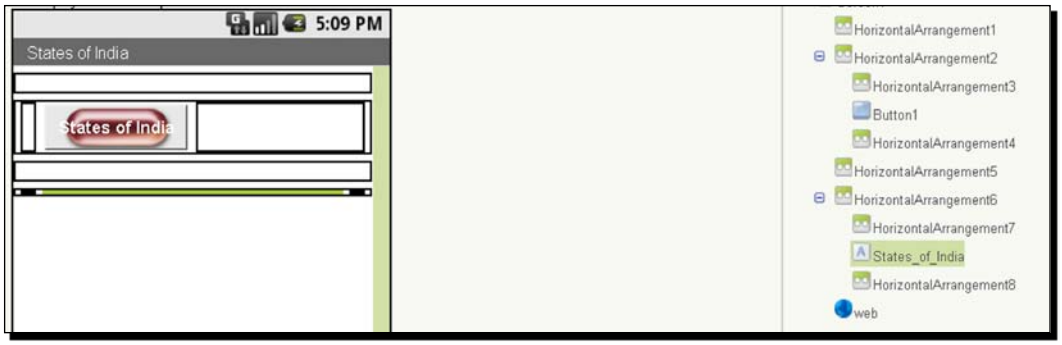
You can get a copy of the following blocks at: `http://arrsoft.com/examples/webtest_parse.zip`



Let's get at it. Make sure your phone or other test device has web access, and away we go!

Time for action – using parsing to break out the states of India

The design part is a breeze—we need only a button entitled States of India, a label (Fill Parent wide, centered) to show the names of the States after they are parsed out, and drop in the **Web** component. Of course, as always, we take a little time to use horizontal arrangements and make everything neat.



And now we go to the **Blocks Editor** and pull together the blocks (as shown earlier).

By the way, be patient. It sometimes takes a few seconds for the web page to load when you tap the button; a lot of stuff coming down.

Here's what we do to build our parser and why we do it:

1. We begin by defining three global variables. We need **x** as a numeric variable (my favourite loop controller), **tmp1** (defined as a text variable), **tmp2** (a temporary list variable)—all three defined as shown earlier.
2. The **Button1.Click** framework gets a **web.Url** block with the address of the Wikipedia page plugged into it, and a **web.Get** block to get the page when the button is tapped. Both those blocks will be found in the **My Blocks**/**web** drawer.
3. Also, from the **web** drawer, drag out that big, bad boy: the **web.GotText** block. Here's where all the magic occurs. Once the web page is downloaded to your phone, this guy takes over.
4. Our first parsing task is to get rid of everything above and everything below the list of States (it's about a third of the way down the page). We do this by examining the source for something unique we can use (precisely the same thing you do in setting up a global search and change in a word processor). I found the string `<li><a href="/wiki/` appearing at the start of the first State and the string **Union Territories** after the last State (we're just doing States in this list).
5. So, we use **split at first** (met in *Chapter 3*, *Playing with Blocks*) from the **Built-in**/**Text** drawer twice to get rid of all above and all below. Note this block puts the results in a list, so we take the second group the first time (letting all above go away) and the first block at the second split, and we're rid of all below. What's left goes into **tmp1**.
6. Since we can see in the HTML source that the string `<li><a href="/wiki/` is in front of every State name (that's part of formatting on the web page), we can use it to break the State names into a list (**tmp2**) using the **split** block (28 states, but still some crud we need to clean out, and will in the final step).

7. Get ready to loop the loop next. Set `x` to `1` and **States_of_India.Text** (our final results label) to an empty string, ready to receive the names of the States.

8. Now, we drag in a **while** loop from the Control drawer. All the rest of our parsing takes place as this loop moves from `1` to `28` (the length of the list of States now in **tmp2**).

I used the **length of list** block (with **tmp2** clicked into it) temporarily plugged into the **States_of_India** label to make sure I was actually accumulating the 28. States into the list. Parsing often requires quite a bit of trial and error to get it right.

9. We now build the **States of India.Text** label—as we loop through each of the 28 items in **tmp2**—by splitting off most of the crud (I found `title` worked for this), and then removing the only two extraneous characters left: an underscore and a trailing quote mark.

10. We increment `x` by `1` so that the next item in the loop gets processed, and when it gets to `28`, we are done.

What just happened?

Out of that mass of web source, we've now pulled a list of all the States of India, looking like this:

Get

Get is an HTTP (Hypertext Transfer Protocol) method for requesting and receiving information from a web server, as is post, a better and more secure method that is not completely working in App Inventor yet. But, using the Get method with AI's **web** component, we can send variables and their data content to servers.

A scripting language is required on the web server to accept and act on a Get request. Python is popular, but the most used language is PHP, most often in conjunction with MySQL (a powerful relational database).

Next, to illustrate how a Get request works, is a very simple PHP script on one of my servers. It simply processes a Get request, recognizing and reading two variables named `one` and `two`. For the purposes of this demo—instead of looking something up in a database—it just echoes the content of the variables back.

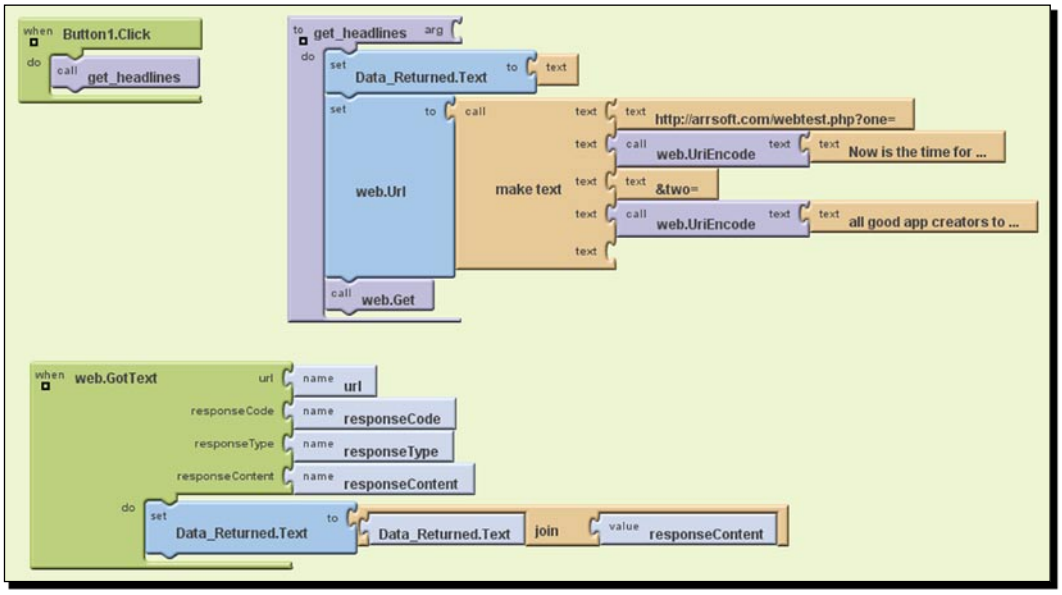
```
arrsoft@taft: ~/html
File Edit Options Buffers Tools Help
<?php

$one = $_GET['one'];
$two = $_GET['two'];

echo $one . "\n\n" . $two;

?>
```

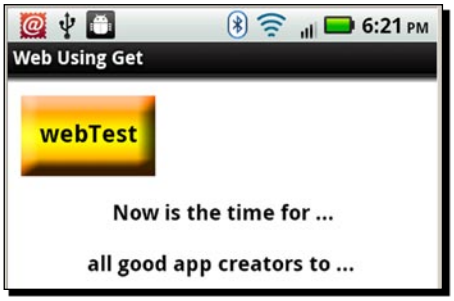
Here are the blocks we use to send the Get request and process the data returned:


when Button1.Click
do call get_headlines
to get_headlines arg
do set Data_Returned.Text to text
set web.Url to call web.Get
make text
text http://arrsoft.com/webtest.php?one=
call web.UriEncode text Now is the time for ...
text &two=
call web.UriEncode text all good app creators to ...
when web.GotText
url name url
responseCode name responseCode
responseType name responseType
responseContent name responseContent
do set Data_Returned.Text to Data_Returned.Text join value responseContent

In our example's design, we need only a button and a label for the returned information.

We create a procedure called **get_data** and call it with the **get_data** block created in the **My Definitions** drawer. In the procedure, we clear our display variable, build a URL (more about that in a moment), and use the **web.Get** block from the **web** drawer to send our Get request to the server and receive the reply.

The server reads the variables with the content specified when the URL was built, and echoes it back, which goes into the **Data_Returned** label and is displayed on the phone, like this:



One thing I hope you noted in the blocks above is that we ran the text going into the two variables through a **web.UriEncode** block. This is necessary because URLs (Universal Resource Locators) or web links cannot have spaces in them, and certain other characters also need to be converted.

While syntax of HTTP requests is beyond the scope of this book, it's something you'll need to know. Just to help you out some, this is the URL our app constructed and sent:

```
http://arrsoft.com/webtest.php?one=Now%20is%20the%20time%20for%20...&
two=all%20good%20app%20creators%20to%20...
```

The `%20`, of course, replaces spaces.

If you have access to a server and a basic knowledge of a scripting language, it's easy to build a quick interface allowing you to send search terms (using SQL or Structured Query Language) and get extensive data back to use in apps, either your own or for other users.

Security is a concern, so you'll want to learn about secure requests. At the very minimum, include a variable with a password in it, and program the interface script on the computer to reject requests that do not include your password.

In your PHP or Python scripts returning data, just have them print a CSV record or table to an otherwise blank web page (no HTML formatting), and the return to **web.GotText** can easily be converted into lists as we did in the first part of this chapter.

If you do not have your own server, you can still use Get requests with servers that allow it (you'll have to figure out the variables they use by doing sample searches online). Usually, these servers will include HTML formatting on the data return page, so you'll have to use parsing on the data, as we did on our list of the States of India.

Summary

In this chapter, we learned the basics of lists and lists of lists and how to use them. We saw how **TinyDB** saves data from session to session, and ways of using the new **Web** component.

Okay, now we have apps that communicate and apps that have data, what's the logical next step? Yeah! The Internet and other networks are there to be used. We create apps using Google's Fusion Tables and also to master the **TinyWebDB** component.

And—now that we're starting to get good at this stuff (creating apps), we'll learn how to put them on the Android Market and other ways of publishing our brainstorms.

7
Apps That Surf the Web

Almost every Android device connects to the Internet. Our apps can also!

This chapter shows us how to use networks and the Internet to use the web and exchange data over the Internet. With the power of apps, you'll find an unlimited data plan on your phone pays for itself soon enough. Let's build some apps that are web-aware!

Here, we will explore examples of:

- Browsing and using websites
- Fusion Tables, Google's free online data service
- Finding a fourth way of online data interchange!
- TinyWebDB, a powerful way to use the Internet to access and store data
- How to publish your apps for the world

We start with browsing and how you can use the millions of dollars invested in constructing huge websites in your own apps.

Browsing and using websites

Back in the day, when I first started writing, there were two ways to research—own a lot of books (I did and still do), and/or spend a lot of time at the library (I did, don't have to now).

This all changed in the early 90s when Sir Tim Berners-Lee invented the World Wide Web (WWW). The web used the Internet, which was already there, although Al Gore did not invent the Internet. Although earlier experiments took place, Vinton Cerf's development of the basic transmission protocols making all we enjoy today possible gives him the title "Father of the Internet".

All of us reading this book use the Internet and the web almost every day. App Inventor itself relies extensively on the web; you have to use the web in designing apps and downloading the **Blocks Editor** to power them up.

Adding web browsing to our apps gives us:

1. Access to literally trillions of web pages (no one knows how many; Google said it was over a trillion in 2008, and growth has been tremendous since then).
2. Lets us leverage the limited resources and storage capacity of the user's phone by many times as we use the millions of dollars invested in server farms (vast collections of interconnected computers) and thousands of web pages on commercial sites (which they want us to use, even beg us to use because it promotes their products or services).
3. Makes it possible to write powerful and useful apps with really little effort on our part.

Take, for example, what I referred to earlier in this book as a link app. This is an app that, when called by the user, simply uses **ActivityStarter** to open a browser and load a web page.

Let's whip one up.

Time for action – building an eBay link app

Yes, eBay loves us—especially if we buy or sell items on this extensive online auction site. And they want you to do it not just at home but when you're out with only your smartphone.

To encourage use from your phone, eBay has spent a gazillion dollars (that's a lot) in providing a mobile site (`http://m.ebay.com`) that makes the entire eBay site (hundreds of thousands, probably millions of pages) available and useful to you.

Back to that word, *leverage*—the ancient Greek mathematician, Archimedes, said about levers, "Give me a place to stand on and I will move the Earth." Our app's leverage won't move the Earth, but we can sure bid on just about everything on it.

Okay, the design of our eBay app is very simple since the screen will be there for only a second or so. I'll explain that in a moment.

So, we need just a label and two non-visual components: **ActivityStarter** and **Clock**. In the **Properties** column for the **ActivityStarter**, put `android.intent.action.VIEW` in the Action field (again, this is how your app calls the phone's web browser, and the way it's entered is case-sensitive).

This gives us something as shown in the following screenshot (with a little formatting added):

Now, the reason for the `LOADING...\nInternet connection` required (in the label, `\n` being a line break) is a basic notifier and error trap. First, if the Internet connection is slow, it lets the user know something is happening. Second, if there is not 3G or Wi-Fi connection, it tells the user why nothing will happen until they get a connection.

Simple additions like the previous (basically thinking of the user's experience in using our apps) define the difference between amateur and professional apps.

Always try to anticipate, because if we leave any way possible at all to screw it up, some user will find it. And who gets blamed? Right. You, me, or whatever idiot wrote that *defective* app. And they will zing you on Android Market.

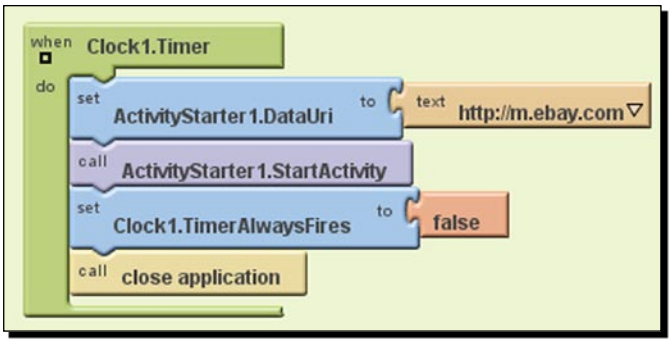
Okay, now to our buddy: the **Blocks Editor**.

We need only one small block group. Everything is in the **Clock1.Timer** block to connect to the eBay mobile site and then, job done, gracefully exit.

Inside the clock timer frame, we put four blocks, which accomplish the following logic:

1. In our **ActivityStarter1.DataUri** goes the web address of the eBay mobile website home: `http://m.ebay.com`.
2. The **ActivityStarter1.StartActivity** block calls the phone's web browser and passes the address to it.
3. **Clock1.TimerAlwaysFires** set to `false` tells AI, "Stop, don't do anything else."

4. Finally, the **close application** block (which we should always be nice and include somewhere in our apps) removes the app from the phone or other Android device's memory, releasing those always scarce resources.

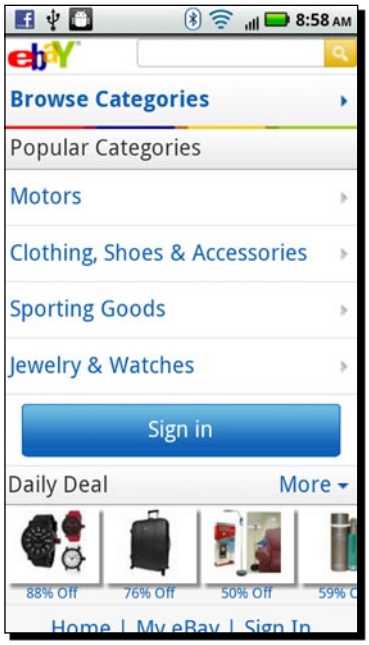


Make a nice icon, and it's ready to go onto Android Market (I put mine on as *eBay for Droids*). And, by end of this chapter, I'll show you how to put App Inventor apps on Market.

What just happened?

That's it—a complete application giving access to a few million great bargains on eBay.

This is how it will look when the eBay mobile site home page opens on the user's phone:



Pretty neat, huh? And there are lots of other major sites out there that offer mobile sites, easily converted into link apps such as this eBay example.

But, what if you want to make the app more complex with a lot of choices? One that after the user finishes browsing a site, they come back to your app and choose yet another site to visit? No problem.

Here's an example of that. I recently published an app called *AVLnews* (AVL is the airport code for Asheville, North Carolina where I live). This app actually lets the user drill down into local news for all 19 counties in the Western North Carolina mountains. Here's what the front page of the app looks like:

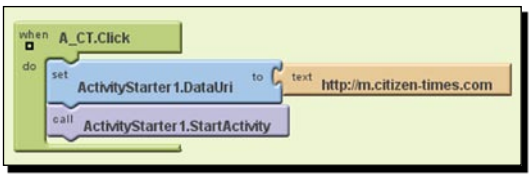
Even with several pages of choices, this type of app is truly easy to create. You need only a few buttons, labels, and (as ever) horizontal and vertical arrangements for formatting. And, of course, add a single **ActivityStarter** for calling the phone's web browser.

Now, here's the cool part! No matter how long and how many pages your user reads on the site a button sends him to, when he or she hits the hardware *Back* button on their phone, they return to your app.

The preceding is the only way the *Back* button works for an App Inventor app! If user is within an AI app and hits the *Back* button, it immediately kills your app (except when a listpicker is displayed, in which case it returns you to the screen that invoked it). This is *doubleplusungood*, so to speak. So, always supply navigation buttons for changing pages and maybe a note somewhere not to use the hardware button.

Okay, back to my *AVLnews* app as an example. The first button I built was for the *Citizen-Times*, our major newspaper in the area. They have a mobile site such as the one we saw earlier for eBay. It looks nice and is easy to read on a Droid or other smartphone, like the following:

And, it's equally easy to program—this is all you need:



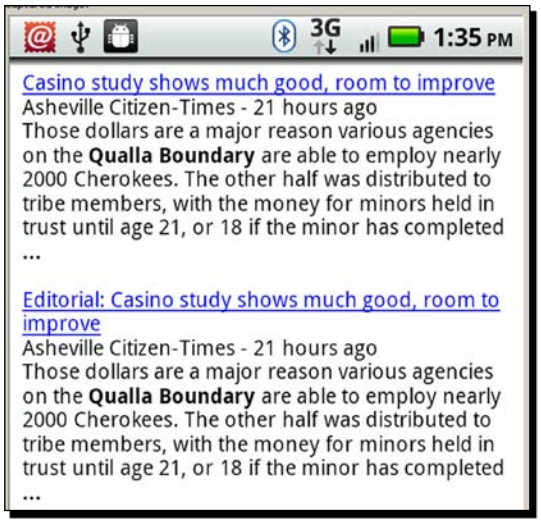
I then moved on to other news sources for the areas. The small weekly newspapers in the more isolated, outlying mountain counties have no expensive mobile sites. The websites they do have are pretty conventional.

But, when I linked to them, the page would come up with tiny, unreadable type. I had to move it around, do the double tap on the content to fit to the page trick to expand the page, turn the phone on its side, and many such more, and still had trouble reading stuff.

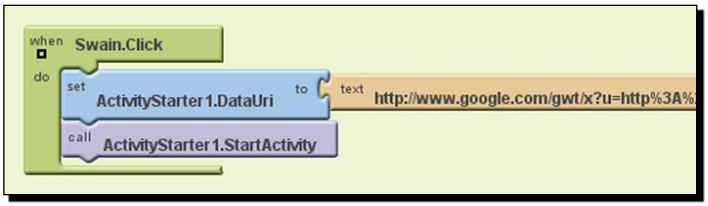
Folks, you do not do things like that to your users. Not if you want them to think you are one cool app inventor, as I know you to be by now.

So, here's the trick to making almost all web pages readable on a phone—Google Mobilizer! It's a free service of Google at `http://www.google.com/gwt/n`. Use it to construct links that return a nicely formatted and readable page as shown in the next screenshot. People will thank you for it and think you expended many hours of genius-level programming effort to achieve it. And, remember, it's not polite to disagree with people, eh?

Download from Wow! eBook <www.wowebook.com>


1:35 PM
Casino study shows much good, room to improve
Asheville Citizen-Times - 21 hours ago
Those dollars are a major reason various agencies on the Qualla Boundary are able to employ nearly 2000 Cherokees. The other half was distributed to tribe members, with the money for minors held in trust until age 21, or 18 if the minor has completed
...
Editorial: Casino study shows much good, room to improve
Asheville Citizen-Times - 21 hours ago
Those dollars are a major reason various agencies on the Qualla Boundary are able to employ nearly 2000 Cherokees. The other half was distributed to tribe members, with the money for minors held in trust until age 21, or 18 if the minor has completed
...

Often, the search terms and *mobilizing* of your link is long and goes on for some time (the following screenshot is only a part of it). However, that's where your real genius comes into play: building a search term that gets the articles fitting whatever is the topic on the button.



Summing up: using the power of the web lets us build powerful apps that use few resources on the phone yet return thousands upon thousands of pages. Enjoy.

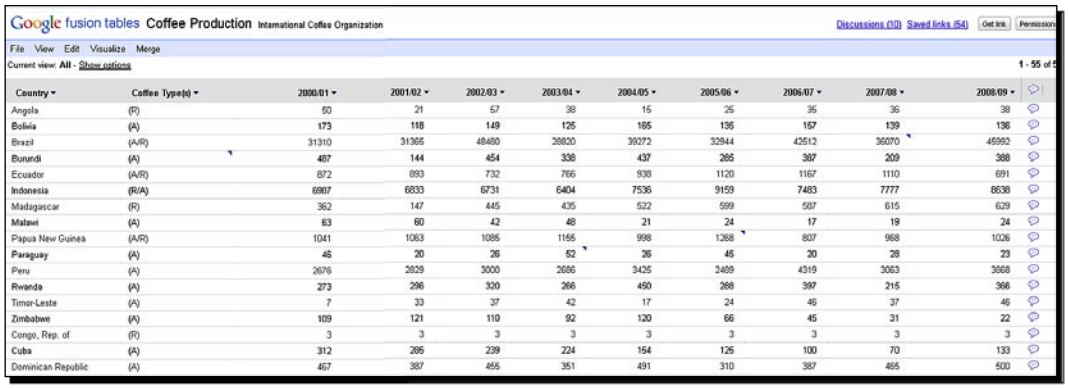
Now, getting back to "Apps that Remember" started in the last chapter, we look at yet another Google service: Fusion Tables. And see how our App Inventor apps can take advantage of these online data sources.

Fusion Tables—Google's free online data service

Google Fusion Tables is a free online service for managing large collections of tabular data (data in tables). If you have a Google account (and you do or you couldn't be using App Inventor), you can upload tables of up to 100 MB and share them with people you choose, or make them public.

You can also import and export Fusion Table data as CSV (comma-separated values).

There are numerous public tables that you can pull data from, such as the following example of coffee imports by country from the International Coffee Organization. I certainly depend on high levels of world coffee production in writing my books.



Google fusion tables **Coffee Production** International Coffee Organization

Discussions (10) Saved links (54) Get link Permissions

File View Edit Visualize Merge

Current view: All - Show options

1 - 55 of 5

Country	Coffee Type(s)	2000/01	2001/02	2002/03	2003/04	2004/05	2005/06	2006/07	2007/08	2008/09
Angola	(R)	60	21	57	38	15	25	35	36	38
Bolivia	(A)	173	118	149	126	185	136	157	139	136
Brazil	(A/R)	31310	31365	48480	28820	39372	32944	42512	36070	45992
Burundi	(A)	487	144	454	338	437	285	387	209	388
Ecuador	(A/R)	872	693	732	766	938	1120	1167	1110	691
Indonesia	(R/A)	6987	6833	6731	6404	7536	9159	7483	7777	8638
Madagascar	(R)	362	147	445	435	522	599	597	615	629
Malawi	(A)	63	60	42	48	21	24	17	19	24
Papua New Guinea	(A/R)	1041	1063	1085	1155	998	1288	807	968	1026
Paraguay	(A)	46	20	26	52	26	45	20	28	23
Peru	(A)	2676	2829	3000	2686	3425	2489	4319	3063	3868
Rwanda	(A)	273	296	320	266	450	288	397	215	366
Timor-Leste	(A)	7	33	37	42	17	24	46	37	46
Zimbabwe	(A)	109	121	110	92	120	66	45	31	22
Congo, Rep. of	(R)	3	3	3	3	3	3	3	3	3
Cuba	(A)	312	295	239	224	154	125	100	70	133
Dominican Republic	(A)	467	387	466	351	491	310	387	465	500

Now, so far, we've mentioned more than once the three main ways AI has to manipulate online data—the **Web** component covered in *Chapter 6*, *Apps That Remember*, Fusion Tables here, and **TinyWebDB** (more about it coming up in the next section).

Having three methods, especially when their functionality overlap somewhat, might be a little confusing. But, again, it's still early days in the development of App Inventor.

Here's my prediction. I think—as the Web component becomes more powerful, especially after headers are fixed so that the Post method can be used easily—**TinyWebDB** (a bit complicated as it is) will eventually fade away. The **FusiontablesControl** component will be further developed and stay around because it's specific to that particular service.

Which of these three methods do you use, and for what?

Web and **TinyWebDB** are better (more flexible, more powerful) if you have access to database frontend (or the ability and permissions needed to program them) on a wide range of web servers.

A second advantage of the previous two ways is the ability to conveniently control data access. If you write apps where online data is provided, for example, you can securely specify what searches are allowed, what, if any, SQL (structured query language) input is accepted and acted on, and so forth. This is typically achieved through a PHP or Python frontend script.

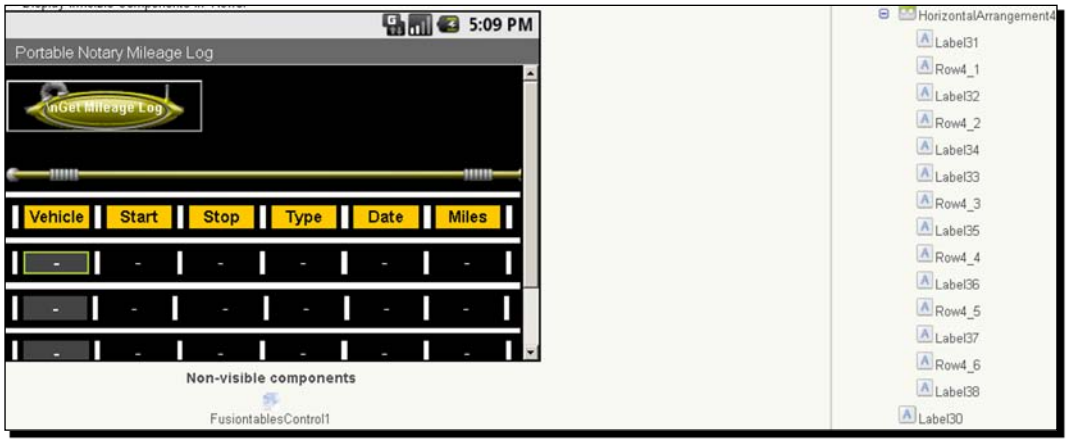
However, certainly the greater majority of App Inventors (we who use App Inventor) will not have extensive access to online servers. In that case, Google Tables is not a bad alternative.

The disadvantages of Google Tables are the limitation in size (yeah, 100 MB may sound like a lot, but databases get *big*) and the difficulty in sharing access to the data.

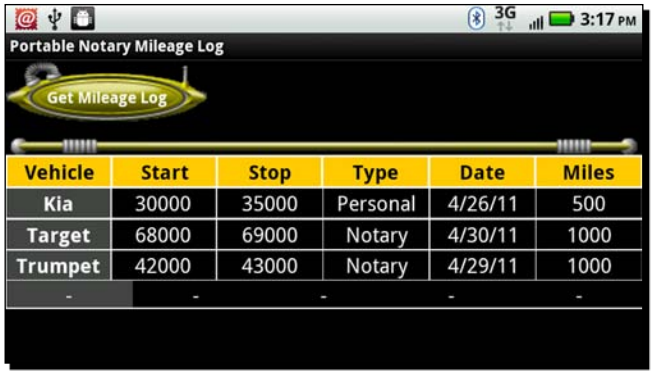
This latter disadvantage is irksome. Google Tables requires the user to be logged in to their Google account to use even the public tables. In writing an app that needs access to online data, you cannot (and should not) depend on your users to all be logged in, especially from a phone.

Anyway, there are apps for your own use that Google Tables will be good for and—since the **GoogletablesControl** component is still under development, we can always hope for a solution to the data sharing drawback (please, guys?).

Earlier I shared with you the mileage log app I'm writing for one of my companies where we need to track travel distance. The following is the design portion of the app:



Constructing a display of table data requires a bit of tedious formatting in order to get a neat looking result as follows. I used labels with a white background and two pixels thick to make all the lines.

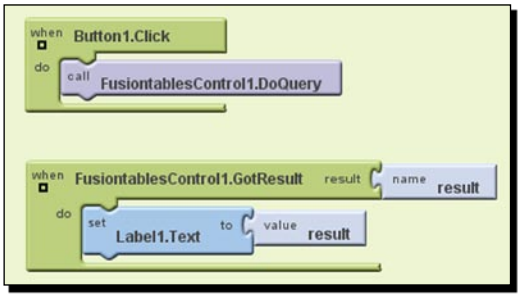


For the **FusiontableControl** component to get data, it has to know which database to query. Luckily, it already knows where Fusion Tables is in general. We get the database number by looking at the `dsrcid` variable in the URL of the database in our browser. The `dscrid` is used to reference a Fusion Table; sort of like an IP address translating to a name. In the case of the `Coffee Import by Country` public table, it is `94782`.

Let's just pull out countries. Go to Google Tables, and look under the public tables for the previous one; it's reasonably close to the top.

So, we make an SQL query using rules and examples in the help material about Fusion Tables online and come up with `select 'Importing Country' from 94782`. We enter this information in the **Query** field that appears in the **Properties** column of **Design** when the **FusiontablesControl** component is in place and selected.

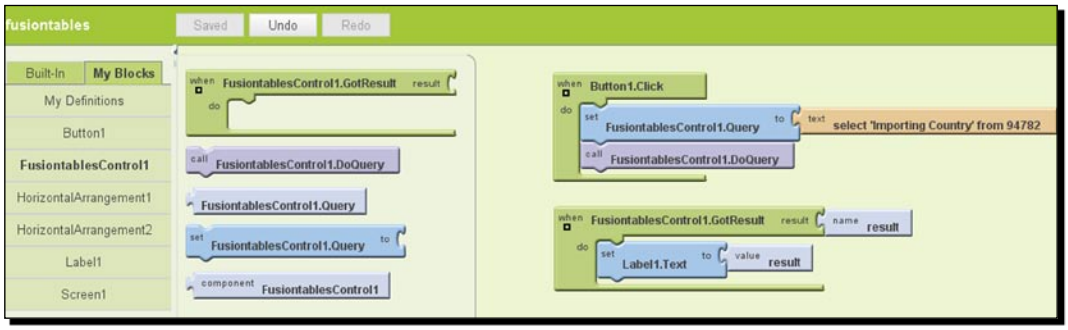
In **Blocks Editor**, we put the **FusiontablesControl1.DoQuery** in the button block (as shown in the next screenshot), which fires off the request when the button is tapped (asking us to log in, of course). And, in the **FusiontablesControl1.GotResult**, we put the value of the result (that block is in **My Definitions**) into a label so we can see it.



And get a list of countries as follows:

We can also use the **FusionTablesControl1.Query** block to let the app control the query to Fusion Tables rather than putting just one into the **Query** field in the **Properties** column of **Design**. This is shown in the next screenshot.

Also shown on the left is the entire drawer created in **My Blocks** when you drop a **FustiontablesControl** component onto the virtual screen in **Design**. Not much to it, which tells us why this component still lives in the **Not ready for prime time** drawer in **Design**.



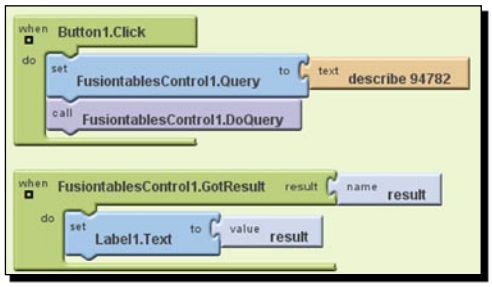
But, this does not mean **FustiontablesControl** is not yet useful. It is. Just so long as we understand the parameters of using Fusion Tables, which include:

1. It's moderately simple, more like a spreadsheet than a powerful relational database such as MySQL, my favorite PostgreSQL, or the commercial powerhouse of Oracle.
2. The search terms are also limited compared to what you can do with the databases listed previously.
3. Despite being able to share your private tables or even make them public, the user is still required to have a Google account to access them; currently a killer so far as using Fusion Tables in widely disseminated apps to the general public, many of whom will not have such an account.

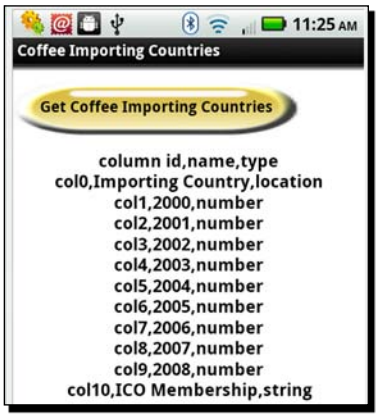
Teaching you how Fusion Tables works is beyond the scope and space available here, but a good starting point in learning how search terms work is: `http://code.google.com/apis/fusiontables/docs/developers_guide.html`.

However, I can (and certainly should) show you what to do with CSV tables imported into App Inventor. This technique works not just with Fusion Tables but with imports from pretty much any online database.

One thing you'll need to know is (as I mentioned several times) the structure of the table you're working with. If you created the table on Fusion Tables, you know its structure. If you want to use one of the public tables, either just take a look at it online or we can use AI to find out for us as shown in the following screenshot by using `describe <number of table>` as the query:



Which, in the case of `94782` (the coffee import table), returns the structure of the table, providing us with a reference to display the CSV table when we import it:



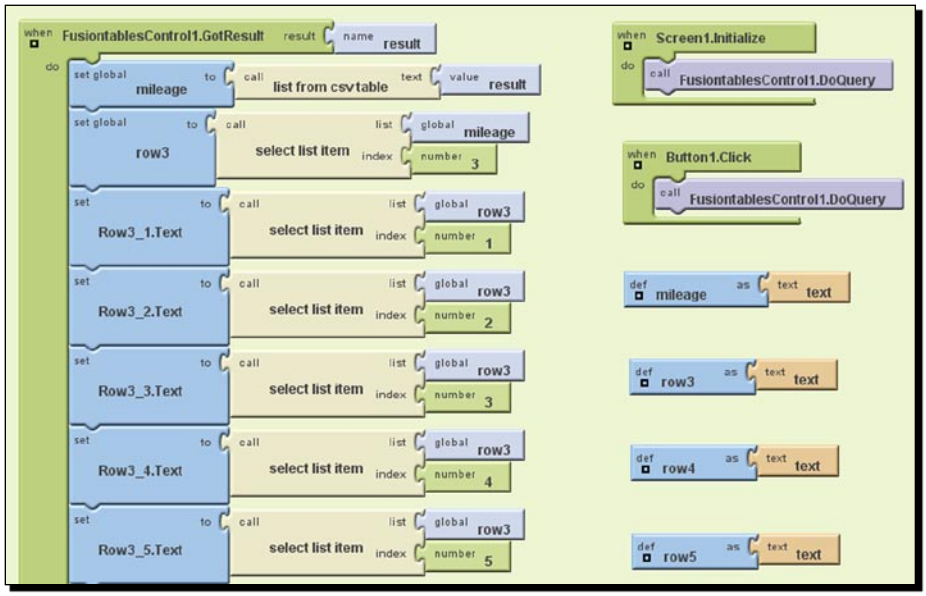
Well, that was easy enough. In the example earlier, we had simple data coming back from our query; that is, not a list, but just a single term (the list of countries) with a handy line return at the end of each one. No formatting was needed in displaying it in a label as previously.

Most usable data from an online database of any sort, not just Fusion Tables, will be in CSV table form—fields separated by commas, making up records separated by line returns.

In the **Blocks Editor** portion of my mileage log shown earlier, using the techniques we learned in the *List and list of lists* section in *Chapter 6*, *Apps That Remember*, we convert the CSV table returned from a query to my Fusion Tables table into lists and those lists into columns and rows for displaying.

The query (search) term I used was `select * from 742086`, which tells Fusion Tables to send us everything in table `742086`—which is a private table of mine.

Displaying a table on the phone is not complicated, but it is a bit tedious. Here's a portion of the way I do it in the `Mileage` app:

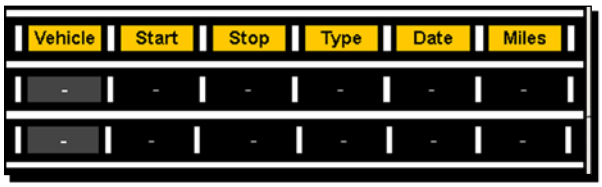

when FusiontablesControl1.GotResult result name result
do set global mileage to call list from csv table text value result
set global row3 to call select list item list global mileage index number 3
set Row3_1.Text to call select list item list global row3 index number 1
set Row3_2.Text to call select list item list global row3 index number 2
set Row3_3.Text to call select list item list global row3 index number 3
set Row3_4.Text to call select list item list global row3 index number 4
set Row3_5.Text to call select list item list global row3 index number 5
when Screen1.Initialize
do call FusiontablesControl1.DoQuery
when Button1.Click
do call FusiontablesControl1.DoQuery
def mileage as text text
def row3 as text text
def row4 as text text
def row5 as text text

The steps are simple enough:

1. When the button is tapped, get the table from Fusion Tables (I also get the table when the app initializes so that the table is populated from the first).
2. Convert `value result` (the table) into the global variable `mileage`. We now have a list of lists where each of the sublists are records (rows) of the table.
3. For each row, select each field and display it. In my case, we have three vehicles for which I want the name of the vehicle, starting mileage, ending mileage, type of trip, date, and current total miles traveled.

To get the correct number of rows (records) and columns (fields), you have to know the structure of the database you're importing. That is critical. Since I created this table in Fusion Tables, I know its structure and, in **Design** (see the next screenshot), can set up a display table with a row of headings (column names), and three rows (one for each vehicle).

I use labels with white backgrounds and two pixels wide as dividers and labels—these latter named as **row3_6** and **row4_5** to show me which row and column they are in. Believe me, in something this (I'll use the word again) tedious, naming your labels is crucial in avoiding confusion.



Tapping the **Get Mileage Log** button imports the table, converts it from CSV to a list of lists, and breaks out the data as shown in the following screenshot. The names of the vehicle (actual names of our three Kias) go in the first column, and the other columns are filled appropriately as to type of trip. Among several businesses we run is a notary signing service doing real estate refinance closings, and that's what generates most trips (see `http://portablenotary.com`). And do check to make sure the app's arithmetic is correct—`35000 - 30000` should be 5000! I need to revisit that app.

Vehicle	Start	Stop	Type	Date	Miles
Kia	30000	35000	Personal	4/26/11	500
Target	68000	69000	Notary	4/30/11	1000
Trumpet	42000	43000	Notary	4/29/11	1000

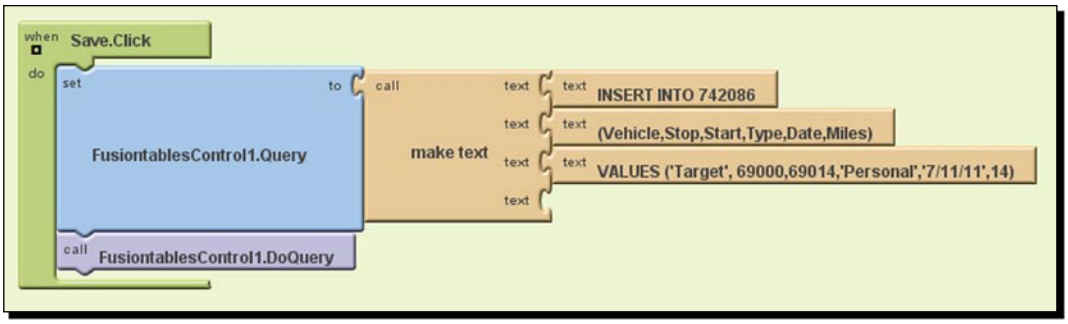
How do we get data back up to Fusion Tables? Well, in the actual production version of Mileage (when I get time to finish writing it), we'll tap on the name of the vehicle (that label will be turned into a button), and a screen comes up showing the last three trips (just for reference) and asking for the new ending mileage and the type of trip.

From that, starting and stopping miles are revised by the app, the type of trip noted, the current date added, and the number of miles for the trip calculated. So, we wind up with a new record (row) of data like this (it can be a list):

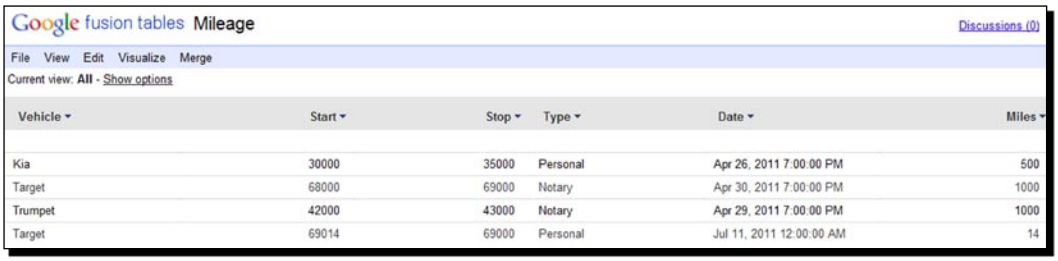
```
(Target 69000 69014 Personal 7/11/11 14)
```

Now, we want to add this new trip for Target (he's my personal chariot) to the table up on Fusion Tables, wherever it be out there in the cloud. The user (me, in this case) simply taps the **Save** button.

Oh, yeah, we better tell that **Save** button how to save. Just to make this faster, I'm going to manually build the query that goes back up (in production, I'll be pulling it out of a list variable to make this same query string). Here's what it looks like:



And, when we check online, we find (last row) it saved the row of data by adding it to the rows (records) that were already there.



Vehicle	Start	Stop	Type	Date	Miles
Kia	30000	35000	Personal	Apr 26, 2011 7:00:00 PM	500
Target	68000	69000	Notary	Apr 30, 2011 7:00:00 PM	1000
Trumpet	42000	43000	Notary	Apr 29, 2011 7:00:00 PM	1000
Target	69014	69000	Personal	Jul 11, 2011 12:00:00 AM	14

That's a basic introduction to Google Fusion Tables and the **FusiontableControl** component.

To sum up, Fusion Tables are a limited but useful way of storing and retrieving data online in spreadsheet-like tables. They have the disadvantage of requiring the user to have a Google account.

And, one more disadvantage: Google Tables are still in beta testing (as is App Inventor itself). Google Tables are slow.

Since Google Tables are slow, you'll want to limit reads and writes as much as possible.

It is now time for us to take a look at **TinyWebDB**—which is the third method of using online databases from App Inventor apps. ... Oops! Wait! I just thought of another way.

Quickie data-getter—finding the fourth way

Okay, here's a bonus, guys. Here, I'm reminded of the old story about the scientists who put a monkey in a room with only three ways out. The monkey finds a fourth way.

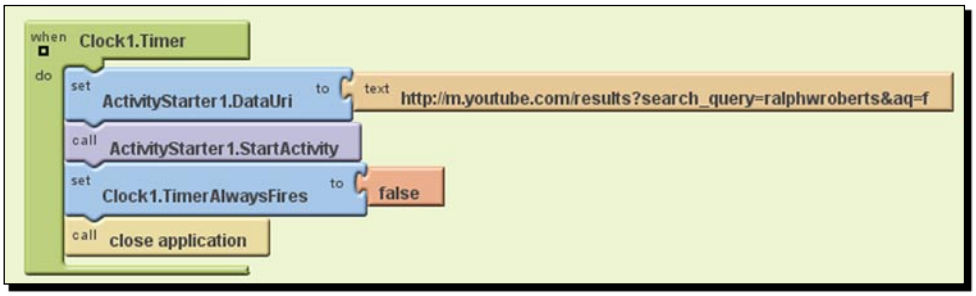
A fourth way of accessing online databases (those that allow Get queries) is a simple **ActivityStarter** activated by a button.

Which gives us this on the phone:

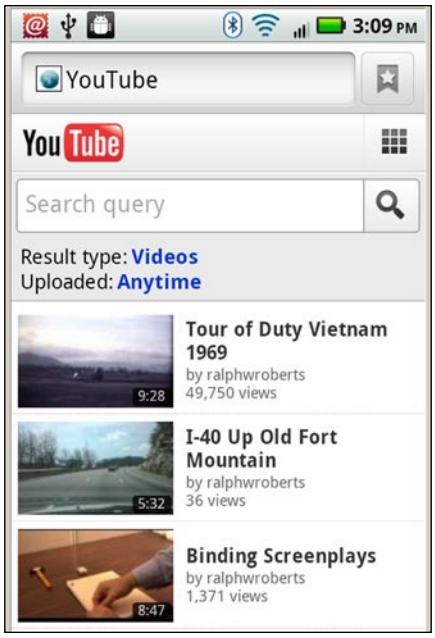
Yes, we did essentially this same thing in the eBay link app example, but we were just calling the mobile website there. Previously, we actually passed a query (via Get) to the server and can pull specific replies from the database.

The advantage to this quickie technique is you don't have to parse the result; it shows up with formatting, images, and so on. For example, I have almost 200 videos on YouTube. What if I wanted a neatly formatted app to help people with Android devices find and watch them easily?

No problem. Just find the right search term on YouTube, and set up my link app as so:



And we get nicely formatted lists of my videos with hardly any work! The previous **close application** block removes the app from memory, once its job of bringing up the video list is complete.



Back, now, to our regularly scheduled chapter in progress.

TinyWebDB—accessing and storing data

Of the three (but keep that fourth in mind as a useful tool) methods to interface with online databases, we find overlap in functions, especially between the **Web** component and the subject of our scrutiny in this section: good old **TinyWebDB**.

Web's the new kid on the block, but if I were a betting man, I would probably have a lot less money, but not because I bet **Web** will eventually replace **TinyWebDB**. Especially when the developers fix that header problem and give us Post that works with **Web**. And that will probably be even before this book hits the streets.

However, at this moment in time, **TinyWebDB** is top dog, database fetching-wise. So, let's get acquainted. Nice doggie.

To start, the emphasis on **TinyWebDB**, as it is with the **TinyDB** component, is on the word *tiny*. Both components were originally developed to make one variable at a time persistent; that is, remembered from session to session. **TinyDB** does it on your phone; **TinyWebDB** on a web server. The advantage of the latter is that this data can be shared between phones since it is available on the web. The ability to share data between phones allows you to create apps to communicate with each other. For example, you can create apps that keep track of highscores.

However, both components essentially save one tag (a name assigned to the variable so that it can be easily retrieved) and one piece of data—be it a number, text variable, or even a variable holding an extensive list of lists—limited in scope. Not the big CSV table chunks we expect when querying a MySQL or one of the other types of large online databases.

But, while **TinyWebDB** was designed for small things, this did not stop several innovative App Inventors from doing some powerful database interfacing with it. Of note is the work done by Shival Wolf, who lives in Australia. Shival—a very helpful regular on the AI forums—has some great examples of **TinyWebDB** using PHP to access MySQL databases and more. Check out his site at `http://ai.kittywolf.net`. Lots of great stuff there!

As stated before, the **Web** component will eventually do what **TinyWebDB** does in interfacing to Internet databases better, but... for now... **TinyWebDB** is what we have.

First, here's the way the designers meant for it to work.

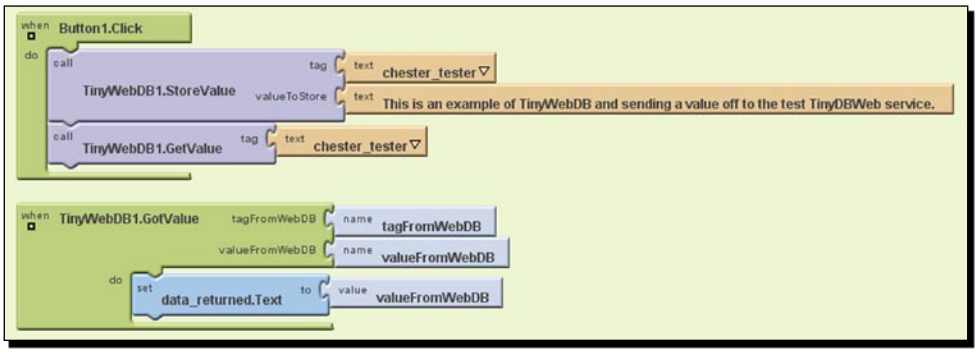
We'll do an example using the **TinyWebDB** test service provided by the App Inventor developers. The link to this server—`http://appinvtinywebdb.appspot.com`—is programmed into the **TinyWebDB** component, and when it is dropped into **Design**, appears in the **ServiceURL** field of the **Properties** column by default. Remember, the data on this service is temporary and should not be used for a production app.

Time for action – testing TinyWebDB

So, let's design a quick test. You need only a button and a label, formatted a bit, naturally, as so:

Download from Wow! eBook <www.wowebook.com>

In the **Blocks Editor**, we'll pull together the blocks below. You could use two buttons—one to store and one to retrieve—but this is more elegant.



Here's what you need and how the previous blocks work:

1. The **Button1.Click** causes the **TinyWebDB1.StoreValue** button to contact the **TinyWebDB** test service online (the URL in the **ServiceURL** field in **Design** previously). It works just like TinyDB in that you have a unique name tag (**chester_tester** for my example; use your own) and a value to store, and that is what happens.

2. The second block in the button framework, **TinyWebDB.GetValue**, immediately read the value just stored back out (using the **chester_tester** tag to call it).

3. To have a way of reading the retrieved value, we drag out the **TinyWebDB1.GotValue** frame (from the **TinyWebDB1** drawer). In it, we put our label text block (**data_returned.Text**) and grab the **valueFromWebDB** block from the **My Definitions** drawer, where AI automatically created it when we dragged the frame block out.

What just happened?

Using our app, we get the following result. Using **TinyWebDB**, our data is written to the **TinyWebDB** test service, pulled back off the web, and displayed on our phone or other test device.

8:02 PM
TinyWebDB
Test TinyWebDB
Get DB
This is an example of TinyWebDB and sending a value off to the test TinyDBWeb service.

Once you get used to AI doing stuff for you like in step three in our previous example, and just accept it, you're well on your way to being able to program just thinking about what you want to achieve without worrying about where your next block's coming from—because you'll already be dragging it into place. It's just like learning a foreign language. One day you're trying to remember memorized vocabulary, the next you're asking for a beer without effort.

I've been learning programming languages (or in the case of App Inventor, ways) for a lot of years now. The point where writing code or moving blocks, while just thinking of the desired results and not the mechanics of it, always sneaks up on me. Once I realize that's what I'm doing, it's always a wonderful feeling!

How do you get there? Simple—use it a lot. You'll accumulate techniques, and pulling blocks around becomes easier to visualize (that's one big advantage of AI, it being a visual language).

Writing *Chapter 3*, *Playing with Blocks* is where I made the real breakthrough. Seeing how all those basic blocks worked together brought the light in making AI much easier to understand and create apps in. I hope it's helped you. If not, hey, check it out again. Get the basics of a language down, and the hard parts become ever so much easier.

Okay, back to **TinyWebDB**. As stated earlier, the word *tiny* applies here, but we can get around that rather elegantly. AI geniuses like Shival have already done it big time, but let me break it down to the very basics—dramatically expanding the power of **TinyWebDB**, that is.

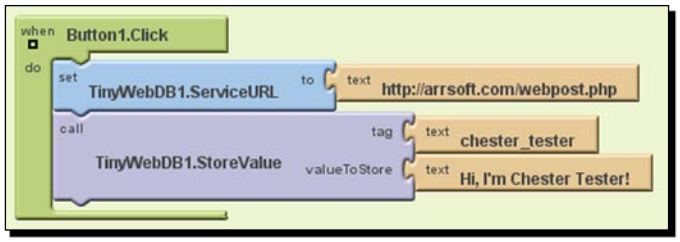
First of all, **TinyWebDB** is limited to a tag and a value pair at one time—that's it. You can follow that with additional operations saving and retrieving other tag/value pairs, but only one set at a time. Exactly the same way **TinyDB** works.

But—and here's the magic—if we use PHP or Python or some other web frontend, we can send SQL commands and whole rafts of varying data at one time and get back tons of data. Let's look at the sending part first.

On one of my servers, I've set up a PHP webpage to receive Post requests from AI (you need access to a server to use **TinyWebDB** or the **Web** component—as we did in the last chapter—this way).

In my app, I use a button (as shown in the next illustration) to:

1. Change the service URL from the default test service to a PHP page on my server, `webpost.php`.
2. Send a tag and value pair to the server using the Post method (the value variable could be an SQL query or records or even tables of data to store in a MySQL or PostgreSQL database).



It is simple to send, and just as simple to receive at the server, as shown in the following screenshot:

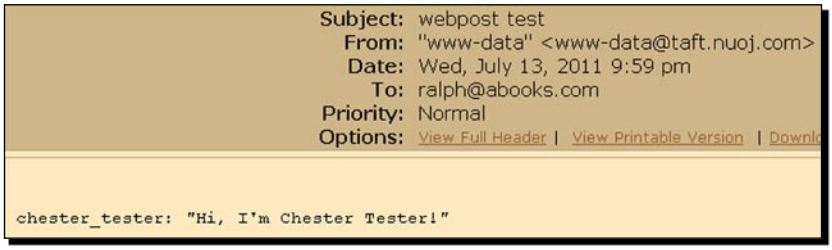
```
<?php

$tag = $_POST["tag"];
$value = $_POST["value"];

mail ("ralph@abooks.com", "webpost test", $tag . ": " . stripslashes($value));
?>
```

The `$tag` and `$value` lines in the previous PHP script receive the name of the tag and the value. Again, the value can be a lot of things and as long as you need. The script can then act on whatever's in the `$value` data received from the app on your phone.

I added a line that sends an e-mail to me just to make sure everything was working and got the following e-mail (which shows the script received and properly processed what **TinyWebDB** sent).

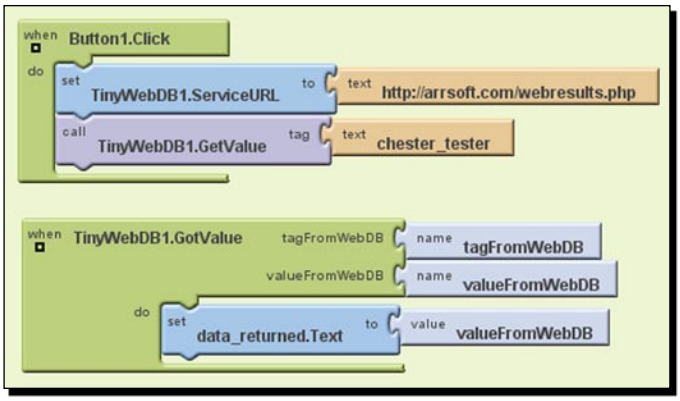


Now, let's get data back.

The `webpost.php` page receives data, so I'm writing one now called `webresults.php` to return data. First, however, let's look at the setup to request data in the **Blocks Editor** (next).

Again, we use a button, and in that framework set the service URL to the results page, `webresults.php`. Then, send the name of the tag we wish to receive (remember, **TinyWebDB** works in terms of tags and values).

The result comes back (packaged and sent to us by the PHP script from the web server). This goes into the **TinyWebDB1.GotValue** block we've already met. The value is put into **data_returned.Text** (a label) to be displayed on the phone.



On the web server, the last two lines of the following PHP script takes the data from the `$tag` and `$value` variables and puts them in an array format that the request from **TinyWebDB** will be expecting and can understand.

A final formatting polish is accomplished by the `json_encode` command, which puts the data into JSON format (JavaScript Object Notation, a lightweight data-interchange format easy for humans and machines to read, write, and parse). The echo command makes it available for **TinyDBWeb** to grab and display it on the phone. More about JSON at `http://json.org`.

```
<?php

$tag = "chester_tester";
$value = "Hi, I'm Chester Tester!";

$results = array("STORED", '"'.$tag.'"', '"'.$value.'"');

echo json_encode($results);

?>
```

And that display looks as follows:

That's a start using **TinyWebDB**. This technique is powerful. All the MySQL or other database manipulation is done by routines written in PHP (or any other web-scripting language) acting on search requests from AI and returning results. The scale of all this can be quite sophisticated, far beyond the simple original tag/value design of **TinyWebDB**.

These routines, means of achieving secure exchange of data, and all that is beyond the scope of this book—which, after all, is a *beginner's* guide.

Also, I think it's pretty much a given that—as we learned in *Chapter 6*, *Apps That Remember*—once the **Web** component is developed a little more, we'll be doing all our online database interfacing using it instead of **TinyWebDB**.

Still, the previous shows you the inherent power of App Inventor and its (now four, eh?) ways of interacting with data on the Internet.

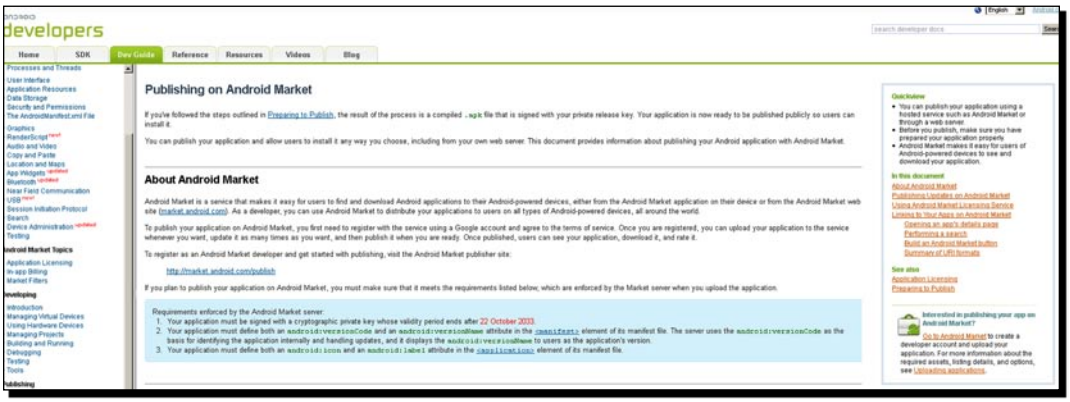
Now for the information I'm sure many of you have been wanting—how to publish your apps on the web!

Publishing your apps for the world

One of the big zings people writing about App Inventor on the web tend to give it is, "there's no way to put App Inventor apps on Android Market." That's totally *wrong*, as I will show you in just a moment. I've got 14 AI apps on there already myself with thousands of downloads. I've even made a little money from them (although most I put on free for you guys to enjoy and learn from).

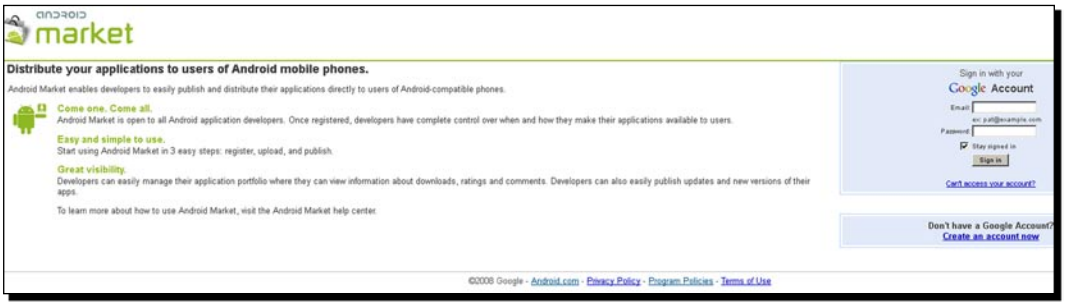
Market, as many of us call it, is the Holy Grail of publishing (and/or selling) apps for Android devices. There are a growing number of other venues—`Amazon.com` being one (and I'm on it, too)—but Market remains the ultimate with tens of millions of folks foraging it for new apps.

Getting access to Market as an app publisher is relatively easy. The site below gives you all the details (`http://developer.android.com/guide/publishing/publishing.html`). There is a 25 dollar per year charge, and if you're selling the app, they take a commission and pay you monthly.



I've seen a number of posts on the App Inventor forums where folk bemoan that 25 dollar charge and look for alternatives. Well, there is absolutely no alternative at this time that gives you what Market does. Access to tens of millions of people for a mere 25 dollar per year is one of those "no brainer" deals.

Once you understand what an app publisher is and how it works, you get a link to the *Developer's Console* (see the following screenshot). Create your account here and log in whenever you want to upload a new app or see how one already there is doing. Numbers of downloads, number of active installs, and various other statistics are provided.



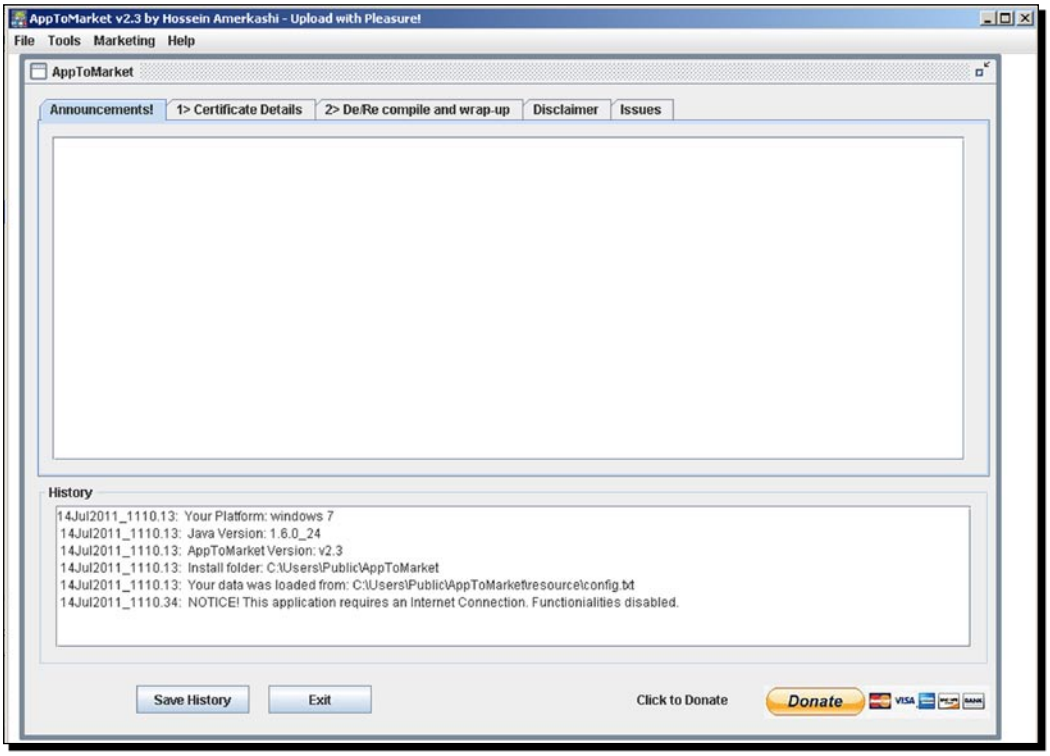
Here's an example of how much exposure you can get in a very short time. Remember the eBay link app we did at the beginning of this chapter? I put it on Market four days ago (at the time of writing), and it's already had over 1,000 downloads and is rated over four stars (see the following screenshot):

To see my apps on Market, just search for `ArrSoft`, my software company name. They're almost all free. And, one thing I do is take pains to provide good-looking `.png` files for all of the suggested artwork (like for my `BigClock` app shown in the next screenshot). Having Photoshop or its free cousin, Gimp, is invaluable for doing these.

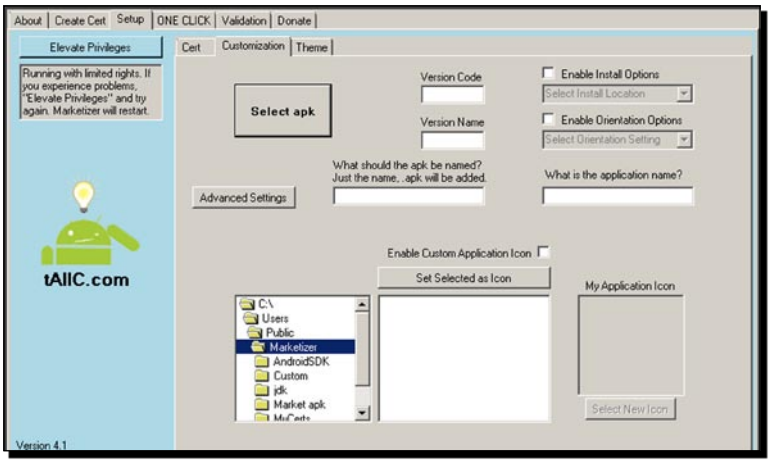
Now, back to the "there's no way to put App Inventor apps on Android Market" mis-statement. True, you cannot *directly* upload an App Inventor `.apk` file (completed app). But, you have three methods of conversion to an acceptable format. You can learn the required format and manually convert AI apps into an uploadable format (painful) or use one of the remaining two options and do the conversion in mere seconds!

Guess which I chose? The easy way, of course. There happens to be two of them by two true App Inventor heroes—Hossein Amerkashi and Gene Kupfer. Both programs work great. You choose.

Hossein's version is called *AppToMarket* and may be downloaded at `http://amerkashi.wordpress.com/` or `https://groups.google.com/forum/#!forum/apptomarket`. It's free to download but supported by donations.



Gene's contribution, *Marketizer*, is supported both by ads and donations. Get it at `http://www.taiic.com/downloads/`.



You'll find one or the other of these two applications indispensible in publishing your apps.

Pop quiz

1. A link app:
    a. Chains your apps together in a logical manner.
    b. Is a quick method of leveraging the power of large websites.
    c. Calls websites.
    d. b and c.
2. Fusion Tables:
    a. Portable power generators utilizing cold fusion that will change the world.
    b. A place to mix ingredients and bond them into one material.
    c. Google's free online service providing "flat" (like spreadsheets) data tables.
    c. The latest furniture style, started last winter in Finland.
3. Using the **TinyWebDB** component, you can:
    a. Save tag/value pairs to a web server.
    b. Send SQL queries to MySQL and PostgreSQL databases.
    c. Request and receive specific data records from online databases.
    d. All of the above.

Summary

In this chapter, we learned how to browse and leverage the power of websites, the use of Fusion Tables (Google's free online data service), a fourth way of online data exchange (using the **ActivityStarter** component), that **TinyWebDB** is a lot more powerful than you would first think, and how to publish our apps to the world.

Now, where were we?

Oh, yeah.

We're getting ready to literally find ourselves in the pages of *Chapter 8*, *Apps That Know Where They Are*. Or at least build apps that know our exact location and how to navigate us to other locations. Throw away your compass (you'll not need that anymore), and here we go.

8
Apps That Know Where They Are

Android devices have a lot of built-in sensors and features that use them. Many of those we've already used, but what about location awareness? Your phone knows where it is, or can find out quickly. And we can build apps using that power!

In this chapter, we explore:

- Changelog, significant changes in the most recent App Inventor release
- Using the location sensor
- GPS and the power above us
- Using Google Maps
- Using the **AccelerometerSensor** Component
- Which way are we? Sideways? Upside down?
- How high are we?
- How far and where is home from here?

First, we look at some new stuff in the latest App Inventor update.

Changelog

As I've mentioned before, it's still early days in the development of App Inventor with updates (we get them automatically, remember) happening about once a month. The previous update—released on June 1, 2011—we covered in *Chapter 5*, *Apps That Communicate*.

July 21, 2011—we got an update (dated as the July 20, 2011 build). This one has a couple of things many of us App inventors have been yearning for—the **Webviewer** component and the ability to use the Post method in the **Web** component. Let's take a quick look at both of those and the other changes in AI before we start creating Apps that know where they are.

Bringing web pages into our apps

As you recall, at the beginning of *Chapter 7*, *Apps That Surf the Web*, we built an eBay link app—that is, an app that does nothing other than use **ActivityStarter** to open a website. My eBay app now has over 4,000 downloads from Market, so many people seem to be finding it useful.

This technique works well and, if you choose a site already formatted for mobile use, looks like a quite powerful app. The big disadvantage being that by turning everything over to the browser, you've lost total control of it.

That's exactly why myself and lots of others wanted a **WebViewer** type component that would import and display web pages inside our apps, keeping the user with us instead of having them wander off somewhere on the web.

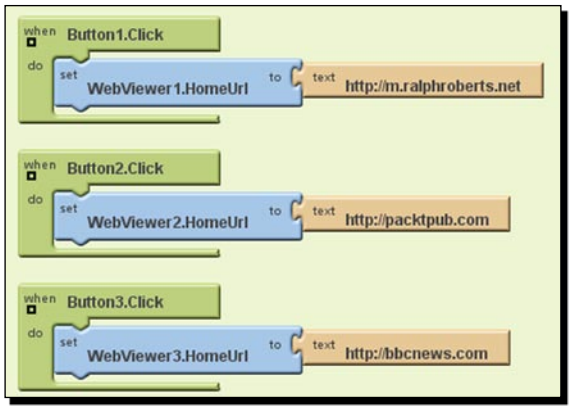
Let me show you an example. We'll be displaying several websites at once under our app's total control.

Time for action – showing three websites at the same time inside an app

First, we need to drop three **WebViewer** components (one for each website) onto the virtual phone screen in **Design**. You'll find this new component in the **Not ready for prime time** drawer in **Design**'s **Palette** column (as shown here):

Set the **WebViewer** components for Fill Parent on width and make them `120` pixels high for this exercise. Also, you'll need three buttons: one for each of the websites.

In the **Blocks Editor**, set up the three buttons as shown in the following illustration using the **WebViewer1.HomeUrl**, and so on. The sites I'm using are the mobile version of my blog, Packt Publishing's (my wonderful publisher), and the BBC News site.


when Button1.Click
do set WebViewer1.HomeUrl to text http://m.ralphroberts.net
when Button2.Click
do set WebViewer2.HomeUrl to text http://packtpub.com
when Button3.Click
do set WebViewer3.HomeUrl to text http://bbcnews.com

I use horizontal arrangements and the **Canvas** component to format and make spacers (`10` pixels high) between the web pages. That's it. Try it out on your phone or other Android test device.

What just happened?

What you should have are the homepages of the three websites you chose displayed as shown in the following screenshot:

Download from Wow! eBook <www.wowebook.com>

Please note, all of these sites are still inside our app! We continue to have control of what the user does, and the user has the freedom of using the web-based information your app provides.

Now, the previous example is not really useful because the user cannot scroll the web pages or navigate the websites.

Here's how you can easily create a powerful and easy-to-use app with the **WebViewer** component.

Have a go hero

You can do it! You can create a neat app with six websites all under the control of your app (that's a really important concept, which is why I've mentioned it three times now) with hardly any effort on your part.

1. First, set up virtual screens like my six-buttons-on-top virtual screen template at `http://arrsoft.com/examples/six_screen_template_buttons_top.zip`.
2. By putting controls (buttons) at the top and making the virtual screens fill parent in width and automatic in height (this is important), the user will be able to scroll websites.
3. Add the **WebViewer1.Url** (or 2 or 3) block for each website you wish to include.
4. And, you'll have an app using the scope and breadth of the World Wide Web, but still under the control of your app.

Using the Post method with the Web component

As mentioned in *Chapter 6*, *Apps That Remember*, the Web component would be pretty awesome if only the developers of App Inventor would fix that header problem preventing it from being used. And, of course, they now have!

First, I revised the small PHP script on one of my servers as shown in the next screenshot. While this is not a book about PHP, I'll throw this tip in for free—using `$_REQUEST` decodes either a Get or a Post method of passing variables to the server.

Again, the advantage of using Post is that it's more secure as the data you're passing is not visible, whereas in a Get it looks something like `http://test.com?one=test&two=also_test`—the latter being more prone to interceptions. Also, you can pass a lot more data, such as entire records or even tables with Post.

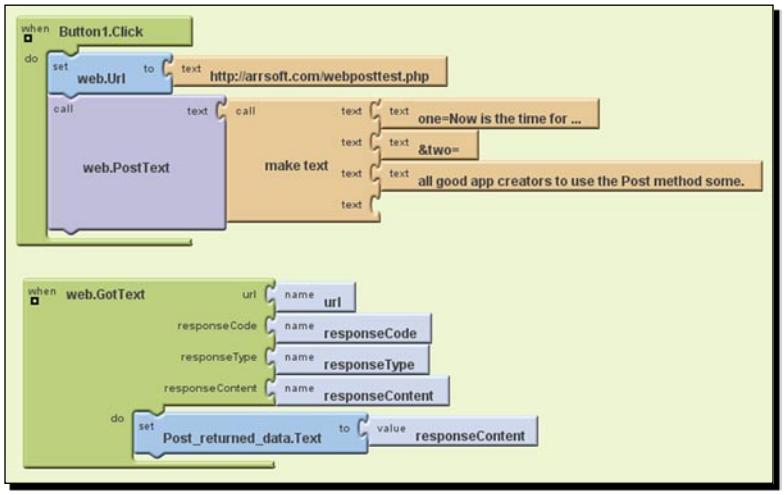
```
<?php

$one = $_REQUEST['one'];
$two = $_REQUEST['two'];

echo $one . "\n\n" . $two;

?>
```

The following is how we set up a Post to our server. Note there is no character in front of the first tag/value pair (or variable and contents of variable or field name and contents of field—all these terms may be used).



In our test (the results on our phone or other test device as shown in the next screenshot), we send data and it is echoed back to use in the preceding **WebGot.Text** frame and displayed in our **Post_returned_data** label.

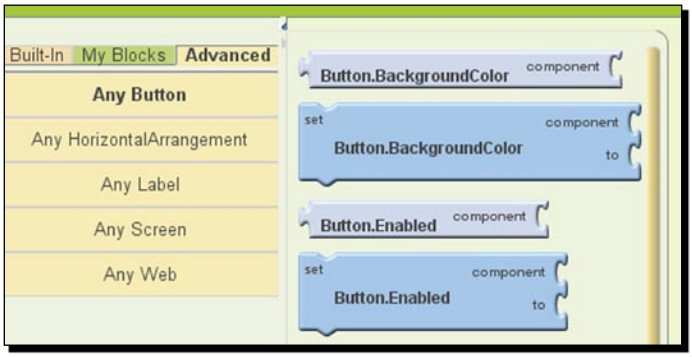
So, Post works mighty fine now! The rest of interacting with MySQL or PostgreSQL, or any other database, can now be done on the server. The result is received back as a CSV record or table, which we've already looked at handling in prior chapters.

If you think **Web** is now a lot easier to do database stuff with than **TinyWebDB**, well... you're right!

Also added to the **Web** component was support (in Gingerbread or above) of cookies, better HTML decoding, and enhanced ways of building Post data requests.

Advanced tab in Blocks Editor

Another change is a new tab, **Advanced**, added in the **Blocks Editor**. It's kind of a sleeper most folks are not catching onto yet since there's little documentation, but it's going to be important. These blocks allow *dynamic* control of components (referring to several of the same type with one block). We'll use this new power in examples to come and show you how it works. Here's how the **Advanced** tab looks:



Think of these as *generic* blocks that work for any component of that type. As we should suspect by now (this being one of the basic ways in which App Inventor works for us), the first time you drop a component of a particular type on the virtual phone screen in **Design**, a drawer with that generic type of component is created in the **Advanced** tab of **Blocks Editor**.

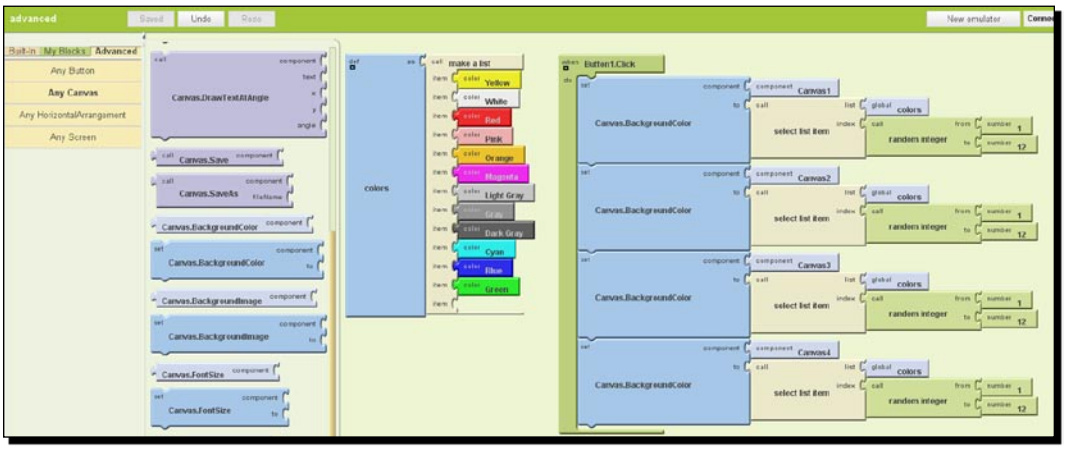
What good are they? Well, we've already mentioned the word *dynamic*—using one reference for several components. This simplifies the number of blocks we need for a given application.

In the following example, I've added four **Canvas** components and a button to **Design** for a little color show.

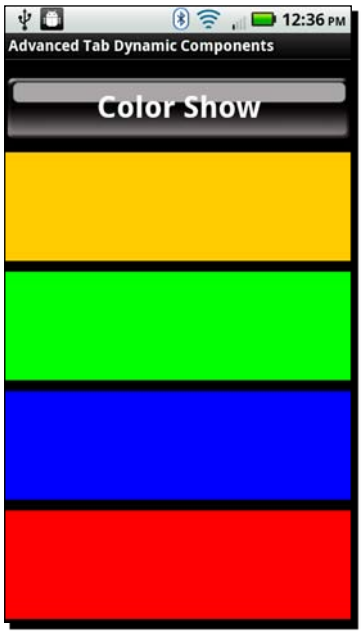
In **Blocks Editor**, I make a list of colors (just the ones from **Built-In**/**Colors** less Black).

In the button frame is where the new magic occurs. To set the color of each canvas, I can use the generic **Canvas.BackgroundColor** block from the **Advanced**/**Any Canvas** drawer (in the next screenshot, that drawer is open).

I set up the first, plugging in the **Canvas1** component block from **My Blocks**/**Canvas1**. The **select list item** (remember, it's a built-in block from **Lists**) gets us a random color. Now, all I have to do is cut and paste this block group and change it for each by only replacing the **Canvas1** block with **Canvas2**, **Canvas3**, and **Canvas4**.



And that's a quick look at dynamic manipulation of components, and my app looks as the following:



Now, here's a brief look at the other changes in this upgrade of App Inventor.

Other changes

Additional new features in the July 20, 2011 build include:

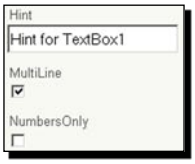
- A new block in the **Text** built-in drawer of **Blocks Editor**, **is text empty?**, which returns `True` if a string is 0—this makes error trapping easier in testing for empty strings.
- Renamed the **Yaw** property of **OrientationSensor** to **Azimuth**, in accordance with changes to Android.
- Added support for Z-layering of sprites. Sprites with higher Z values are drawn in front of ones with lower Z values—in other words, animation sprites now work in three dimensions and can now also move back and forth between layers in addition to up and down and from side to side! This is not to be confused with a true 3D z-axis, but is more like Photoshop layers.
- Added **Reset connections** to the **Connect to Device** menu in **Blocks Editor**, which does an *adb* (Android Debug Bridge) restart (*adb* being the underlying software on your computer making the use of **Blocks Editor** possible).
- The **Blocks Editor** no longer needs to be open in order to package your app. This refers only to packaging on your computer; **Blocks Editor** is still necessary to install on a phone or other Android device attached to your computer.

There were a few more minor things and also a number of bug fixes in this release. By the time this book is in your hand, you can count on several more releases and great new features. By that time, we all should be up to speed in AI and able to grab and use the new stuff as it is introduced.

Son of Changelog

So, this morning, just a handful of days after the last release, there's yet another App Inventor release (dated July 25, 2011). It's mostly bug fixes, but there are a couple of neat new features on the **TextBox** component I need to mention.

First, is the **Multiline** property for **TextBox**. You'll find it in the **Properties** column in **Design** whenever a **TextBox** is selected, as shown in the next screenshot. In the associated drawer created in the **My Blocks** tab in the **Blocks Editor**, there's also a block now letting us set the number of lines we wish the textbox to display. This is something quite a few people have asked the developers for.



Also, something else has been added that fixes a problem annoying many, including me. Before this new block, also in the **TextBox** drawer in **Blocks Editor**, when the user tapped the textbox and entered some text and clicked a button to save it or otherwise do something with the input, the phone's pop-up keyboard remained in place. Now, just add the block below and it gracefully hides itself. Nice to have!

Now, where were we again? Oh, yeah, apps that know where they are.

Using the location sensor

Among the several features that make smartphones so powerful are onboard sensors—those being devices telling the phone various things about its environment.

The three main built-in device sensors (accessed by components in **Design**'s **Sensor** drawer) that we can use in our AI apps are:

1. **AccelerometerSensor**: allows our apps to determine and perform tasks based on how fast the device is moving.
2. **LocationSensor**: lets the app know where the device is located and do things related to that data.
3. **OrientationSensor**: vertical, horizontal, tilted, and so on—our AI apps know in what orientation the device is being held and can process and use that data also.

We'll look at using the location sensor first.

There are a couple of exercises that help us better understand the limitations in accuracy and places where decent results can be achieved.

Here are the limiting factors and two basic methods of the location sensor's operation:

1. **Network**: using the `network` method, the location sensor triangulates (a way of calculating position from two or more sources) from the nearest cell towers available. If there's only one, the result might be off by hundreds of yards/meters or even further.
2. **GPS**: accessing available Global Positioning Satellites with the device's built-in GPS antenna, the location sensor under good conditions can locate the device to within 10-20 feet anywhere on Earth.

Ah, sounds like we should be using GPS for location, eh?

Here's where reality intrudes. If you are in a building or in dense trees or a mountain blocks the device's view of the sky, then GPS signals can't be received.

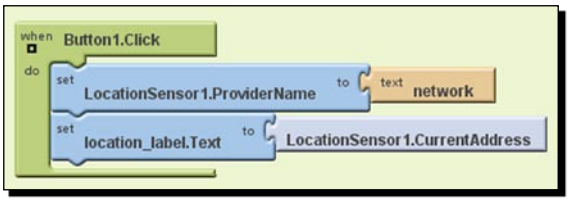
Using Google Maps

But, basically, the most reliably accessible method is network, and GPS the most accurate. Let's test this for ourselves.

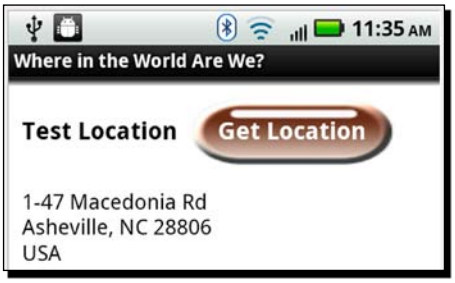
Time for action – determining our location by network

Drop a button and a **LocationSensor** component onto the virtual phone screen in **Design**.

In the Blocks Editor, set up the button as shown in the next screenshot. Install this app on your phone or other test device so that you can take it outside and test also.



Tap the button to see where the phone thinks you are. Here's mine:



What just happened?

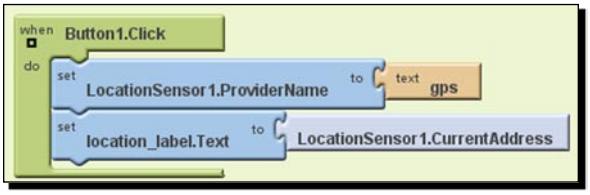
Well, I hope it was more accurate for you. I live out in the country, and we have a weak signal from just one tower here. Still, the location sensor tried its best and did not do too bad actually—only missed the correct town and zip code (postal code) by about a mile.

If, however, one assumes 28806 came out this far, the street address (based on that extrapolation) would be right. So, the network physical location seems pretty close, but it fell down on translating that into the right street address (more on that in a moment).

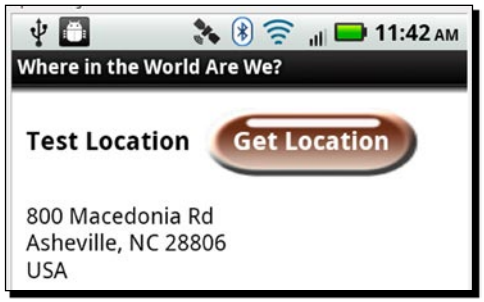
Next, we try GPS.

Time for action – determining our location by GPS

All you need to do is change `network` to `gps` in the following screenshot (be sure to use all lowercase; uppercase does not work) and reinstall your app. Now, get right next to a window or go outside.



Out in the yard, I tap the button and now get this result:



What just happened?

Well, that's kind of right. The location is spot on, the zip code is still wrong, and the town is wrong, but there are mitigating circumstances.

The county we're in thinks our address is the previous one except in 28804 zip code. We're actually in 28701, Alexander, North Carolina. When the county changed our address a few years ago (for emergency services reasons), we fought it because our business address would change and have cost us thousands of dollars.

In short, we won and got to keep 65 Macedonia Road, Alexander NC as our mailing address and compromised on the emergency services address. I tell you guys this because a) it's still a pleasing triumph to me over local bureaucrats and b) our warehouse and store officially has two addresses. And, yeah, you can't expect the GPS to know stuff like that.

Anyway, as you've no doubt found out in your own tests, GPS is more accurate than network.

As they say, *Seeing is Believing*, so let's take an actual look at ourselves from space and see how close GPS gets us using a simple link app:

Time for action – taking a look from space

Start a new app in **Design**. I'm calling mine `from_Space`.

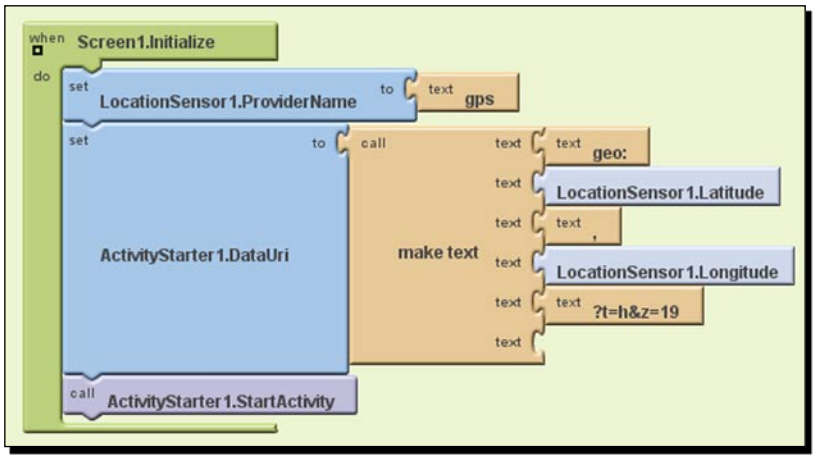
Drop in a button, a **LocationSensor** component, and an **ActivityStarter**.

With the **ActivityStarter** selected, fill in the top three fields in the **Properties** column as follows:

- **Action**: android.intent.action.VIEW
- **ActivityClass**: com.google.android.maps.MapsActivity
- **ActivityPackage**: com.google.android.apps.maps

Now, we go to the Blocks Editor.

This is just a link app (that is, it opens Google map satellite view in our phone's browser), so we put everything in the **Screen1.Initialize** frame block.



The **ActivityStarter** parameters we specified on the **Design** page builds the URL to Google Maps using the latitude and longitude found by the location sensor as in the blocks we built previously.

The result opens our phone's browser and shows us the satellite view of our current location as mine currently in the next screenshot. The blue arrow shows my current location in my home (well, it's about 15 feet off, but that's pretty darn close from space). The grayed out circle is the estimated accuracy of the GPS fix (which again is not bad since it's from inside the house).

Just for your information, the larger building above the house is the bookstore and warehouse of my publishing company, aBooks, and the white car parked there (a Kia Sorento) is my personal vehicle. As you can see, I have a long walk to work, eh?

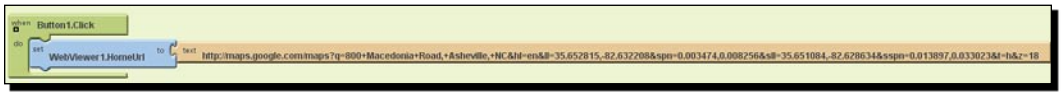
What just happened?

By combining the power of App Inventor with that of Google Maps, we can build some very powerful-looking and useful apps that in fact are quite simple to create.

You can also do the previous backwards by punching the street address of various locations into `http://maps.google.com` and copying the results (upper-right corner of the website, clicking on the little chain link gives you the actual link of that map).

Now, you can take those map location links and use buttons and **WebViewer** components to show them inside your app. As stressed earlier, this allows us to control the user's experience rather than giving it up to the phone's browser.

Here is all you need in the button to open a map in a **Webviewer**:



And it might look something like this:

1:46 PM
Looking at Ya from Space
Activate Space Camera
800 Macedoı
Macedonia Rd
DigitalGlobe, GeoEye, Map data ©2011 Google - Terms of Use

Note the presence of our button at the top. We can also add other buttons and maintain control!

Yes, **WebViewer** is cool and the power of Google Maps awesome, but what else is possible with the sensors in our phones?

Using the AccelerometerSensor component

Using the **AccelerometerSensor** component, we can determine how the device running our app is moving in three different dimensions (its orientation). The way 3D works in our phones and in most modeling and animation software is by using combinations of three directions:

- `x`: side to side, left or right
- `z`: up and down
- `y`: forward (toward you) and back (away from you)

As we saw in the *Changelog* section at the beginning of this chapter, animated sprites in App Inventor now have the `z` axis, thus allowing them to move back and forth in addition to up and down and from side to side or in three dimensions!

A bit more about speed and acceleration—*speed* is the measure of how fast an object is moving at any point in time, while *acceleration* is the measure of how fast an object changes speed over a period of time (its *velocity*). Acceleration on our phones and generally in science is measured in meters per second squared.

Meters per second squared is the result from plugging acceleration and time into the *acceleration formula* (look it up on Google; we don't want to get too far into high school physics here).

If we drop a bowling ball off of a high tower, it starts at 0 velocity and drops at a rate reaching 9.8 meters per second or 32.2 feet per second (that being an acceleration of *one gravity* here on Earth) for an average distance of 4.9 meters in the first second. In the next second its average velocity reaches 14.7 meters and covers 14.7 meters for a total of 19.6 meters for the two elapsed seconds, and so on until it crunches into the sidewalk going really fast.

And before someone thinks to remind me of my high school physics, that actually only applies exactly if the ball is dropped in a vacuum. In the real world, it reaches a *terminal velocity* where the friction of passing through the air overcomes acceleration, and it maintains a constant speed but still really fast when dropped from a height.

But, that's enough about dimensions and acceleration to let us understand how the acceleration sensor on our phone works, and what the following experiment in App Inventor shows is the sophistication of that sensor.

Time for action – seeing gravity and acceleration on our phones

Okay, let's look at acceleration in three dimensions on our phones using the **AccelerometerSensor** component.

All you need in **Design** is a label to show results. Center it, and make it fill parent. Also big enough to see as you swing the phone back and forth through the air (carefully, carefully, of course).

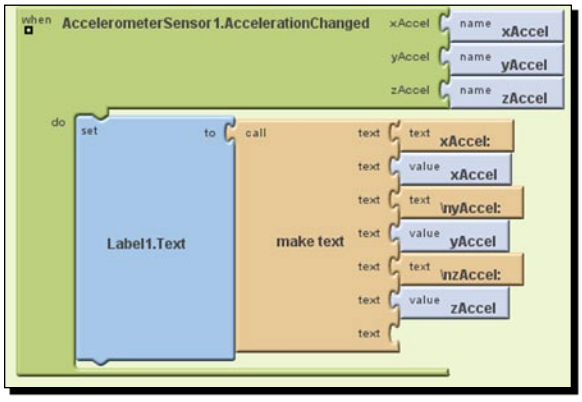
Your design should look something like the following screenshot. I used a horizontal arrangement `15` pixels high to space the text down a bit and make it look neat and made the centered text `32` in size.

From the **Sensors** drawer, drag out an **AccelerometerSensor** and drop on the virtual phone screen.

Move over to the **Blocks Editor**. Pull out (from the **My Blocks**/**AccelerometerSensor1** drawer) the **AccelerometerSensor1.AccelerationChanged** frame block.

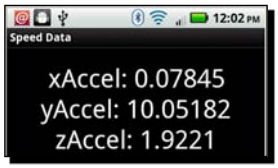
Into it, as shown, get the **Label1.Text** block from the **Label1** drawer in **My Blocks**. Click in a **make.text** block from **Built-In**/**Text** (these are my favorite blocks for constructing strings). Remember to add `\n` to cause each value to be displayed on a separate line for ease in reading.

The rest is easy. The text variable blocks in the next illustration contain labels (so we know which dimension is being shown—`x`, `y`, or `z`. Put the value blocks (automatically created for you in the **My Blocks**/**My Definitions** drawer; just pull them out and click into place) between the labels.



What just happened?

On your phone, you should have all three accelerations displaying as below. Please note that the `y` acceleration is somewhere close to `9.8` because even though the device is not moving, it's still being acted on by gravity—potential acceleration.



Now, pick up your phone and move it around. You'll see the three acceleration values change as the device moves in each of the three dimensions.

Acceleration, when moving in more than one axis, is called vector acceleration, and that starts getting more complicated than we want in an overview book like this. But, all this is important and used by the phone at a lower level than AI can currently manipulate to do such simple things as rotate the screen when the phone is turned from portrait to landscape orientation.

And while we can't change the acceleration sensor readings, we can use them. Here's a more practical use of the **AccelerometerSensor** component.

Time for action – nice and level

Let's turn our phone or tablet computer into a virtual spirit level (the real ones have a little bubble of air floating back and forth) like we use to make sure shelves and the like are level.

The source file for this app is at `http://arrsoft.com/examples/level.zip`.

In **Design**, we need a label for our app, an exit button to make certain the sensor shuts down nicely when the app is no longer required, a promo button if you like, the same label we used to show sensor output previously (but this one will just be for one axis), a canvas, an animated sprite, and a line to show the center of our spirit level. Also, add an **AccelerometerSensor** component and an **ActivityStarter** component.

Also, click on **Screen1** and set the **ScreenOrientation** in **Properties** to Landscape as we want the longest edge of the device to serve as the edge of our level for greater accuracy.

How to Center: Time for a reminder of a very useful technique using the **HorizontalArrangement** component, *centering*. In the previous **Design** screen, we need to center the vertical canvas (which serves as a hairline for our level). Center like this:

1. Start with a horizontal arrangement. Make it Fill Parent in width and Automatic height.
2. Put two horizontal arrangements inside the first. Both should be set Fill Parent/Fill Parent.
3. Place the object to be centered between the two internal horizontal arrangements. It's now centered—an easy and useful technique.

Now, back to our design for the level—here are the important parts of our design, and why:

4. Use an **Exit** button (just drop in a close application button in the button frame as shown in the following source). This is more important even than normal because the acceleration sensor is a resource hog, and we want to make sure it gets turned off when not in use.
5. For the "bar" our virtual spirit bubble runs in, use a **Canvas** component that's Fill Parent wide and `40` pixels high. I made my screen background black and this canvas orange, just for looks.
6. For the "bubble", you can drag a **Ball** from the **Animation** drawer. Make its radius `20` pixels, which will be `40` pixels in diameter, and the ball will be the same height as the bar it "floats" in.

7. Beneath the bar, we need our center "hairline" (use a **Canvas** component), which is done using the previous centering technique. Make it `5` pixels in width. I used a white background; that is, the default color for a canvas.

8. Those are the critical elements. Format the rest of the screen as pleases you.

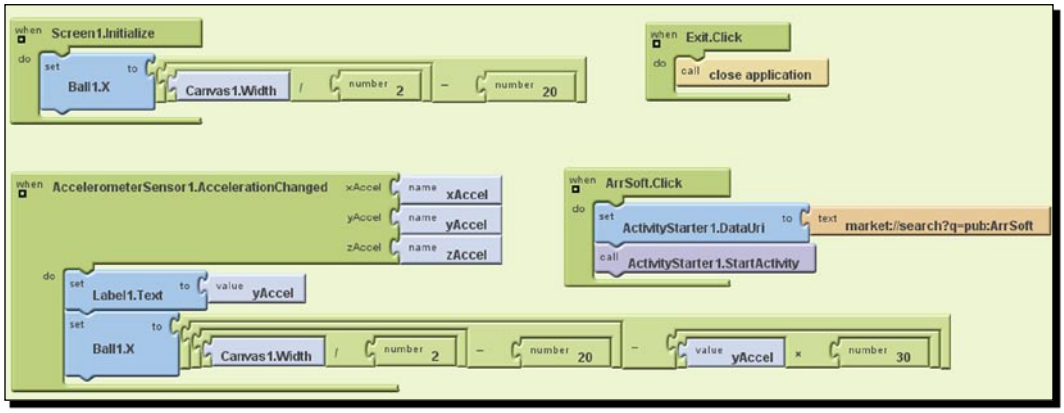
Okay, we now take the train over to the **Blocks Editor**.

The blocks needed to enable our level are pretty simple. Here's what we need:

9. In the **Screen1.Initialize** block, we want to cause the ball (our "bubble") to be centered when the app is started. We use the **Canvas1.Width** block (**Canvas1** is our bar) to get the width of the device's screen, divide that by two, and subtract `20` (half the ball's diameter, because otherwise the edge of the ball would be centered).

10. The **Exit** button gracefully closes the app as described previously and shown in the next screenshot.

11. The **Arrsoft** button is my promo button; make your own or feel free to use mine.


when Screen1.Initialize
do set Ball1.X to Canvas1.Width / number 2 - number 20
when Exit.Click
do call close application
when AccelerometerSensor1.AccelerationChanged
xAccel name xAccel
yAccel name yAccel
zAccel name zAccel
do set Label1.Text to value yAccel
set Ball1.X to Canvas1.Width / number 2 - number 20 - value yAccel × number 30
when ArrSoft.Click
do set ActivityStarter1.DataUri to text market://search?q=pub:ArrSoft
call ActivityStarter1.StartActivity

12. That leaves the **AccelerometerSensor1.AccelerationChanged** frame where all the real operation of the level occurs. We use the `y` axis (up and down) because when the phone is rotated to landscape, `y` becomes `x` (side to side). So, plug the value **yAccel** block (automatically created; get if from the **My Definitions** drawer) into the label that displays about the level bar.

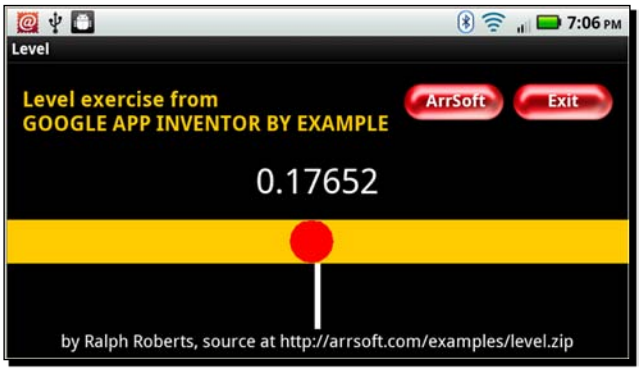
13. Now, use the same blocks we used to initially center the ball/bubble, but subtract **yAccel** from it times `30` (see the previous blocks for how this is accomplished).

What just happened?

Okay, now the level works! (See the next screenshot for its appearance on a phone).

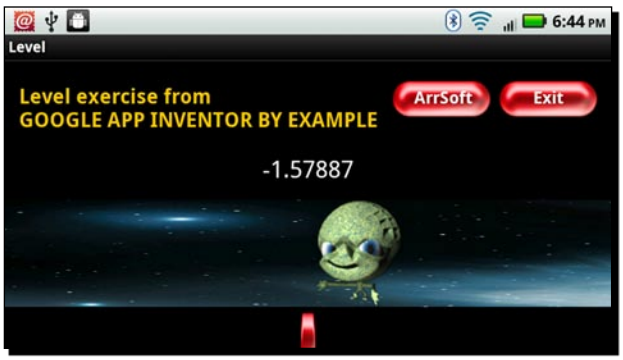
The way it works is as we move the phone around with its edge or back on a plane (such as a table or shelf to be leveled), the `y` axis (now the `x` axis because of rotation to landscape mode) is multiplied by 30 (arbitrary number to make the display smoother by scaling up a bit).

This value is subtracted from the ball's centered position, with a negative causing it to go right and a positive left. The label shows the value and goes to `0` when centered, meaning the surface is level.



Have a go hero – make a more realistic level

It's easy to make a more realistic-looking level by using an **ImageSprite** component instead of the ball and putting an image background on the canvases, such as my *Alien Level* here.



Which way are we?

Okay, the preceding sections were all about the **LocationSensor** component, which gives us data about our location from either nearby cell towers (network) or satellites (GPS), and the **AccelerometerSensor** component, which measures acceleration (movement in the `x`, `y`, and `z` axes). Now, we play with the third type of sensor in your phone that the **OrientationSensor** components give us access to.

The raw data from the **OrientationSensor** is in degrees, the type of degrees in angles and compass headings. And, by the way, that's exactly what we are getting. Here are the three streams of data on orientation this component provides us:

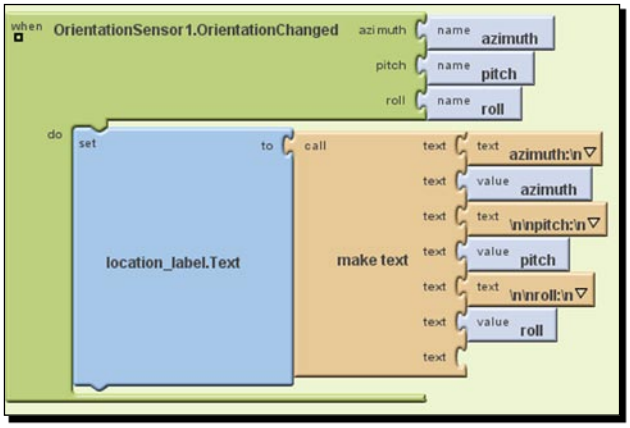
- **Azimuth**: A compass heading where `0` degrees is due north, `90` due east, `180` South, and `270` west for a total of `360` degrees.
- **Pitch**: Hold your phone straight up in portrait mode. Now, pretend you're flying an airplane (it's okay to make airplane engine noises as you do this). Tilt the phone forward and tilt it backwards—this is pitch, as you dive bomb the alien invaders and pull out at the last moment. Pitch angles are in degrees: `0` to `360`.
- **Roll**: Tilt your phone from side to side. That's *roll*. Roll is in degrees, just like pitch.

Let's see what it looks like as we fly ... er, experiment with orientation.

Time for action – seeing azimuth, pitch, and roll

In **Design**, drop in a label. Give it center justification, and make it about 32 points in size. Drop in an **OrientationSensor** component.

In **Blocks Editor**, drag out the **OrientationSensor1.OrientationChanged** block. It's all we need.



Download from Wow! eBook <www.wowebook.com>

As ever, we find that App Inventor has ever so nicely created the value variables we need in the **My Blocks**/**My Definitions** drawer. Use those to build your label as previously, and check the result on your phone or other test device.

What just happened?

We now get a running account of the phone's orientation as to compass heading azimuth, pitch, and roll. Move it around and watch the results (as shown below):

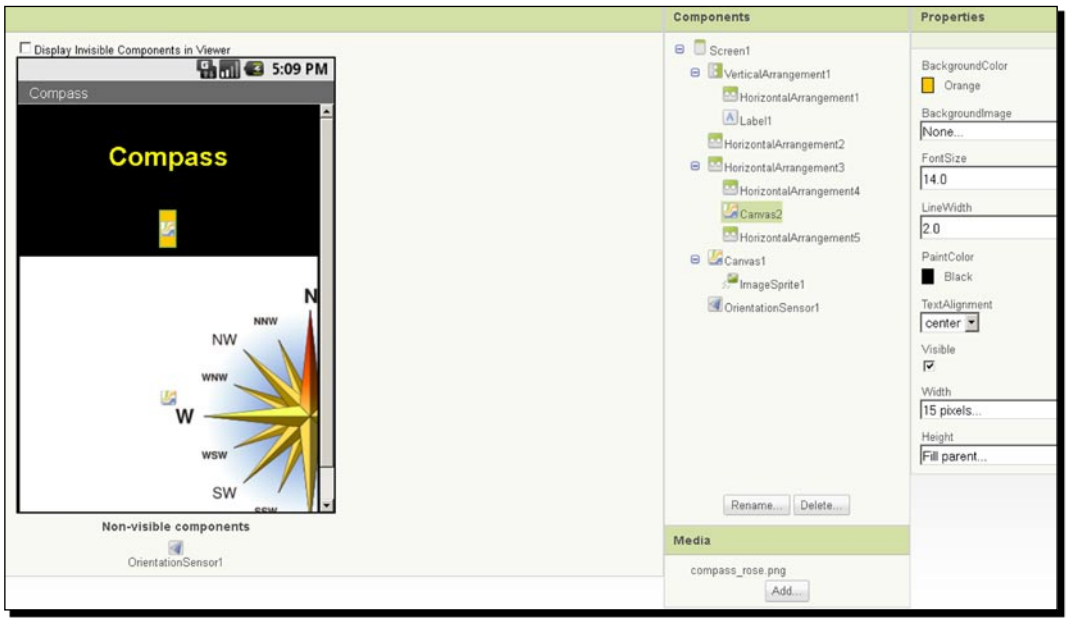
Want to make something useful with this orientation data? What about a compass?

Time for action – building a compass

The layout for our compass (shown in the next screenshot) is simple (okay, the whole thing is simple—again, the beauty of creating apps with App Inventor). In **Design**, the important components are an indicator (the small canvas centered as we did earlier), a large canvas, (mine is Fill Parent wide, `300` pixels high), and an image sprite on top of the large canvas.

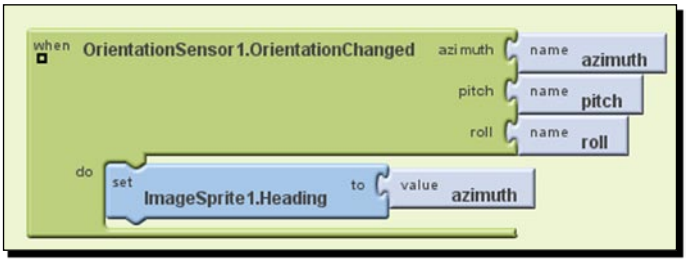
For the image sprite, I found the graphic of a compass rose (the twirly thingamabob in compasses) and let that be the image for the sprite, making it `300` pixels high and wide.

Drop in an orientation sensor, and we're done with design!



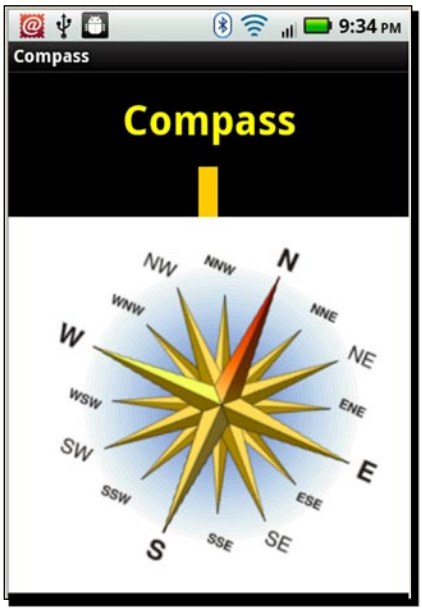
And, the neat thing—looking at my test device—I saw that the compass rose sprite was already centered! So, that really makes things a breeze as we saunter over to the **Blocks Editor**.

Drag out the **OrientationSensor1.OrientationChanged** frame block from its drawer. Get an **ImageSprite1.Heading** (heading is another term for azimuth) and click it in, and a **value azimuth** from **My Definitions**. I love the magic of all these powerful blocks automatically created and waiting for us based on what we chose in **Design**. Done here.



What just happened?

We just built a compass with practically no effort. The compass rose rotates around just as in a real magnetic compass, as shown in the following screenshot. Is that cool or what?



Now, just how high are we?

How high are we?

I live in the mountains of Western North Carolina where driving only a mile or two can dramatically change the altitude (the height above sea level).

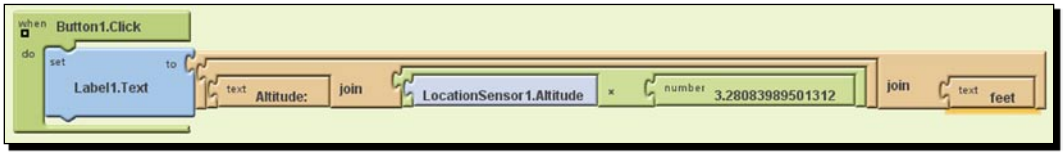
I often like to know my altitude and (being in the U.S.) get it in feet instead of meters as is the default. You might also like to know. We go back to the **LocationSensor** component for that.

Time for action – finding your current altitude

All you need in **Design** is a button, a label, and drop in a **LocationSensor** component.

Most of the effort, in my case at least, goes into converting meters into feet (that's the 3.2808... conversion number). If you live in a country that uses meters, it's even easier.

Set it up in **Blocks Editor** as follows:



What just happened?

With a mere tap on a button now (assuming we have a good view of the sky and our phone's GPS radio is receiving satellite signals), we get our current altitude.

So, we know things now like our location, the compass direction we face, but just how far from home are we? Let's build an app that knows.

How far from home are we?

That network of Global Positioning Satellites (GPS) up there in the sky is a wonderful resource. Mega-millions of bucks were invested in those just so we can use them for free. So... *let's*.

Got a really neat app for you called **How Far Is Home?**, and we're going to build it now.

The most tedious (but rewarding) part of it is designing the user interface. I've put the source for the entire app on the web so that you can download it and follow along easier. Get it at:

`http://arrsoft.com/examples/HowFarIsHome.zip`

Before we start, I'd like to give a callout to Professor David Wolber, one of my App Inventor heroes. Dave got me started in determining how to calculate the distance between two GPS locations.

Distance between two geographic points (latitude and longitude) is the basis of our app. We set our home address (or any other address we want to do a running calculation of distance in miles to). Thanks to Dave's tip, we find the formula for it along with an explanation of how it works at: `http://www.meridianworlddata.com/Distance-Calculation.asp`.

We begin, as always, in **Design**. The necessary components are a lot less complicated than a first glance (shown in the next screenshot) might lead you to believe. Basically, all we need on the screen are three labels and two buttons. We also include three non-visible components, which get used as so:

- **LocationSensor**: Used to get the street address, the latitude, and the longitude of our current position via GPS.

- **Notifier**: Serves as a safety feature in preventing inadvertent overwriting of the Home address. Also tells us when the save function of a new Home address succeeded.
- **TinyDB**: Allows us to save our Home address as well as its latitude and longitude from session to session (make it persistent data).

It really is worth taking pains in setting up any app you create—makes you look good! Use the downloaded source of mine to see how I use all those horizontal arrangements to space and position components.

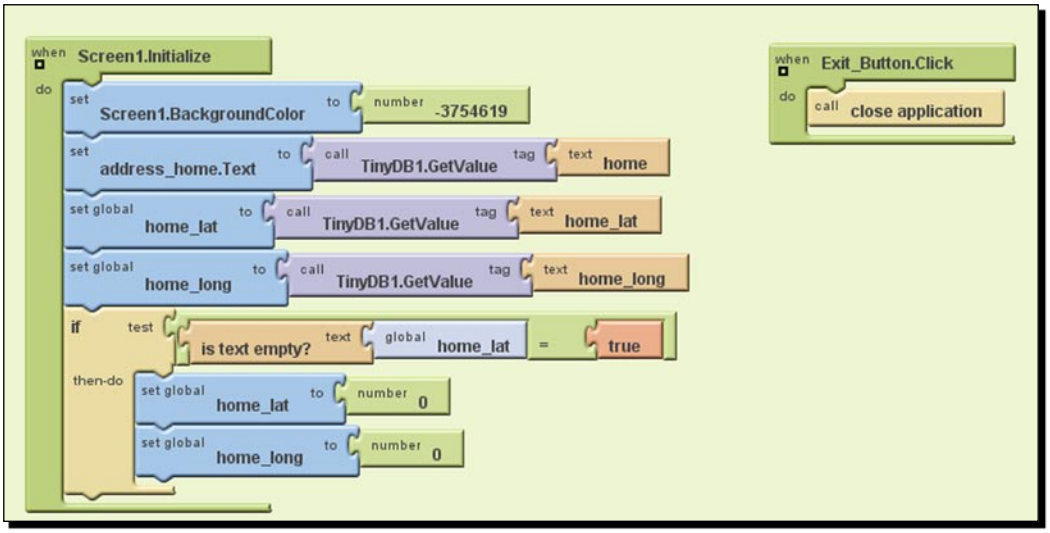
Several general considerations in the design of this app might prove of beneficial. Or at least give you an introduction to what you have to think of up front.

- Put the most viewed things at the top. In this case, that would be the changing distance to home and the current address.
- Some things you can't help, so point them out prominently. GPS is sometimes slow and requires line of sight to the sky for the satellites to connect with a usable signal. I have two notes about that—that it does take time and that you have to be outside. Otherwise, people blame you if they are in the basement of an old bomb shelter under 400 feet of concrete and are not getting a GPS signal. Go figure, huh?

- If there is data people probably want to keep, such as their home address, make sure they can't easily overwrite it, or they will. That's why we use the **Notifier** component to catch accidental hits on the **Set Home** button.
- Make operating buttons stand out, easy to find, and easy to tap.
- Always present the user with a clean, neat design.

As to the latter bullet point, I spent most of the morning on designing the interface and maybe an hour or so getting the blocks to work right. And that's the way it should be, I think.

Now, to the **Blocks Editor**—let's look at it in segments. First, the beginning and the end:


when Screen1.Initialize
do
set Screen1.BackgroundColor to number -3754619
set address_home.Text to call TinyDB1.GetValue tag text home
set global home_lat to call TinyDB1.GetValue tag text home_lat
set global home_long to call TinyDB1.GetValue tag text home_long
if test is text empty? text global home_lat = true
then-do
set global home_lat to number 0
set global home_long to number 0
when Exit_Button.Click
do
call close application

Time for action – creating the end and the beginning

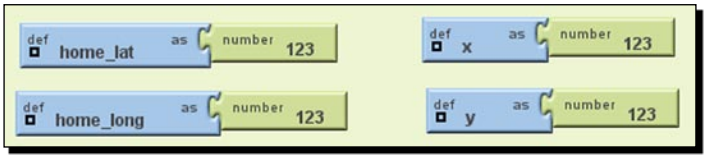
Looking at the previous screenshot, the first thing we build is the **Screen1.Initialize** frame (get it from the **My Blocks**/**Screen1** drawer) block group. It's what runs whenever the app starts up. In this block, we accomplish the following (see if you can find the blocks as we go):

1. Set the background screen color. I use the random color generator we built earlier in the book. Instead of the 13 colors that come standard with App Inventor, we have our choice of literally hundreds of thousands! I chose `-3754619`, just because it looked good to me and made my app design a little bit more unique than those usually seen.
2. The next three blocks retrieve the stored **TinyDB** values for our home address as well as its latitude and longitude. In just a moment, we'll see how those values get in there.

3. In the **If then-do** block, we error trap by filling empty values with a `0` for latitude and longitude (as in when we start the app for the first time and no home location has ever been stored). This prevents a crash later on, and I'll show you where.

While we're at the beginning, we also need to declare four global values (shown in the next illustration). This can be done before or after the initialization block; App Inventor does not care, it just wants them there.

We'll need global variables for the home latitude and longitude and for **x** and **y**, which will be used in calculating the distance from home.



Finally, we drag out the **Exit_Button.Click** frame and drop a close application component in it so that the app can exit gracefully, removing itself from memory. For any app that streams resources as this sensor-based one that's constantly updating GPS coordinates, closing it right for the user is a *very* nice (and professional) thing to do.

What just happened?

Our initialization tasks are now completed, and we added the proper exit button. Now, we find out where we are.

Time for action – getting the current address, longitude, and latitude

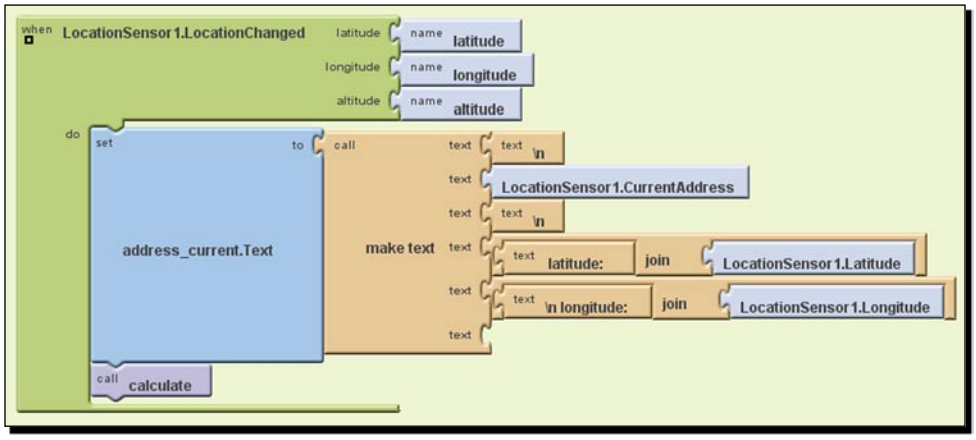
To get the current location continuously, we choose the **LocationSensor1.LocationChanged** frame from the **My Blocks/LocationSensor1** drawer. This frame automatically generates name (local) variables for latitude, longitude, and altitude in the **My Blocks/My Definitions** drawer. However, we will not use the name variables—we just want this frame because it fires off every time our location changes. This constant updating is quite nice.

As we see below, we use the **address_current.Text** block to build and display our current location in the—wait for it, eh?—**Current Location** window on the phone's screen.

We start with a line return just for formatting and add the blocks (also from the **My Blocks/LocationSensor1** drawer) containing the data we want. That being what the GPS system *thinks* is our current street address. It is a lot less accurate in guessing the address than it is with the latitude and longitude.

GPS, when you're getting good signals from several satellites, is somewhere around 20 feet or so in accuracy—which is pretty darned good considering how big this planet is.

The address and coordinates also tend to drift around a bit even when the phone is just lying on your desk. Accept that; nothing else comes that close. Certainly it's better than asking some guy on the corner who gives incoherent directions. And, his guess at how far you are from home is sure to be way off. We'll look at the calculate procedure next to see how that's handled.

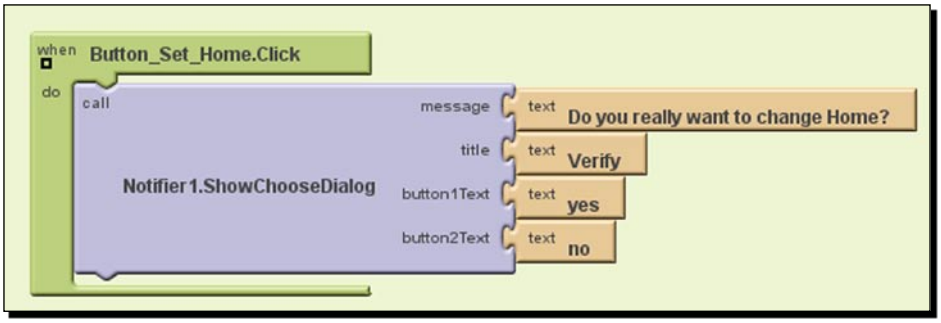


What just happened?

The phone, under our app's kind guidance, has now gotten and displayed our current location.

Time for action – saving our home location

To have a home address, sometime you'll have to actually go home and save the current location. The **Button_Set_Home.Click** frame activates the **Notifier** component as so:

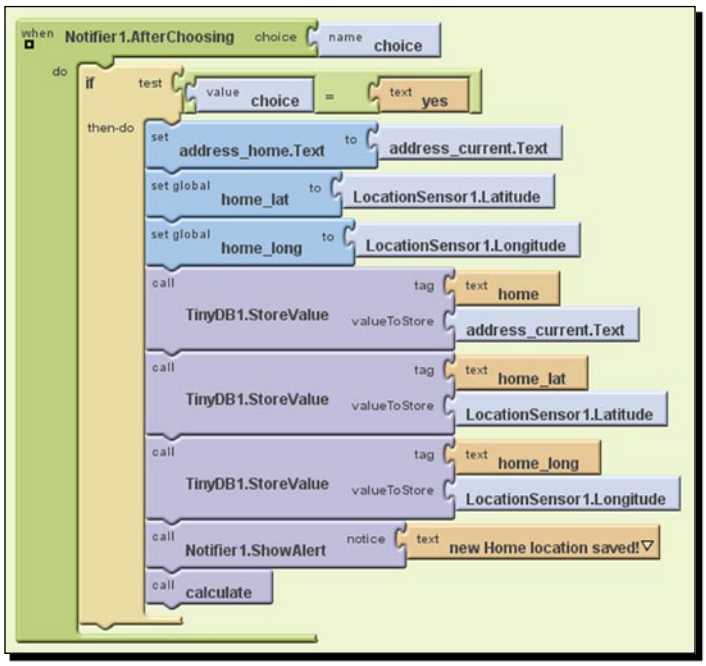


We now do some real work in the **Notifer1.AfterChoosing** frame (**My Blocks**/**Notifier** drawer).

The only choice we care about is *yes*. Choosing *no* just causes the notifier box to go away, and the home address does not get overwritten (the whole purpose of this safeguard).

If *yes*, we put the current address into the **address_home.Text** label, and it displays as our home address in the **Home Address** window on the phone. We also put the current latitude and longitude into the **home_lat** and **home_long global** variables (need that for calculations shortly).

Then we save all that data into **TinyDB** so that we'll have it the next time the app starts. We then notify the user (by flashing an alert on the screen) that the save was successful. And, finally, we call the **calculate** procedure because we're now ready to find out how far we are from home.



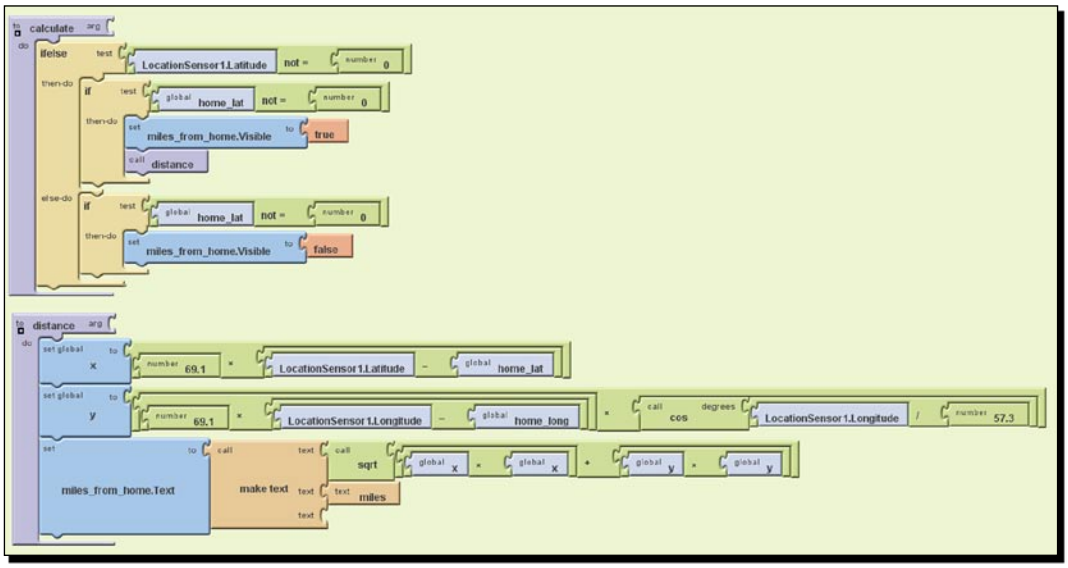
What just happened?

We just saved our home location, so now we can go places, and our app will calculate how far away we are.

Time for action – the distance between home and wherever here is

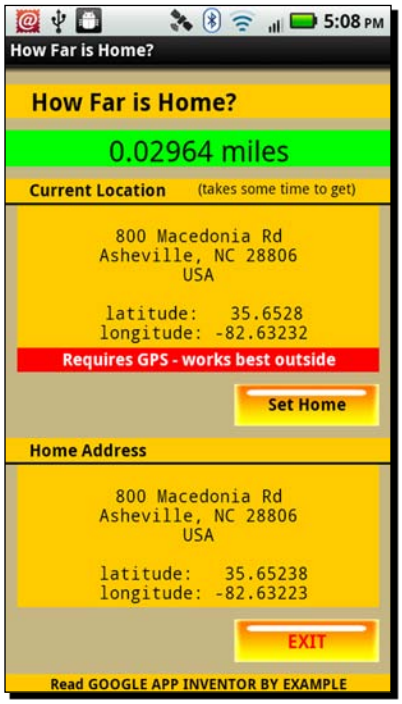
To find our distance from home, we need to turn those formulae on `http://www.meridianworlddata.com/Distance-Calculation.asp` into blocks (as shown in the next screenshot). I'm using the second (more accurate) variation of the formula.

The **calculate** procedure is just an error trap to make sure the distance is calculated only when both a current and a home set of latitude and longitude are present. If they are, the **distance** procedure is called and the distance is calculated, the word miles added, and displayed.


to calculate arg
do
ifelse test LocationSensor1.Latitude not = number 0
then-do
if test global home_lat not = number 0
then-do
set miles_from_home.Visible to true
call distance
else-do
if test global home_lat not = number 0
then-do
set miles_from_home.Visible to false
to distance arg
do
set global x to number 69.1 × LocationSensor1.Latitude − global home_lat
set global y to number 69.1 × LocationSensor1.Longitude − global home_long × call cos degrees LocationSensor1.Longitude / number 57.3
set miles_from_home.Text to call make text text call sqrt global x × global x + global y × global y
text miles
text

What just happened?

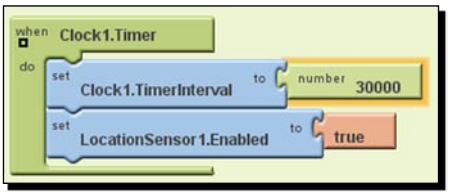
Our app does well now in finding the distance between our home (or any other place we've set) and our current location. See it in the following action. But, the app is not quite ready for prime time yet. You can fix that.



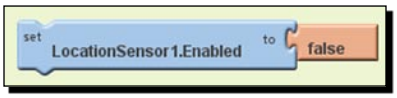
Have a go hero – turn the GPS On and Off

I took this app out for a drive this evening (well, my wife was driving). Since the GPS is always running, it will eventually crash. The solution, I believe, is to turn the GPS off sometimes. You can do that by adding the **Clock** component.

Once you have a **Clock**, pull out (in **Blocks Editor**) a **Clock1.Timer** frame. This will cause the GPS to turn on depending on when the interval is set. I'm trying `30000` now; it seems to be okay.



The final thing, as the last block in the **distance** procedure, is add the following block. After the distance is calculated, this turns off the GPS until the clock timer turns it back on.



I also added a button for instant calculations (turns on GPS and calls the **distance** procedure).

Summary

In this chapter, we met the recent significant changes to App Inventor via the latest two releases, enjoying especially the new **WebView** component and the addition to the **Web** component allowing us to finally use the Post method in accessing online databases.

We then played with the location sensor and used GPS and the power above us in all those satellites, tying all of it into using Google Maps.

The **AccelerometerSensor** component let us turn things sideways, up, and down, and quantize that into data. We also built a pretty neat compass to go with the virtual level we also crafted in this chapter.

We found how high we were in altitude and also how far we were from home.

So, that's it for apps that know where they are and all that good motion-related stuff.

We've had fun so far, but it was all learning how and then building practical stuff. All that Android power just laying there; why, you'd think someone would play games with it…. And we *will* in the next chapter!

9
Games and Animation!

Google App Inventor has all we need to construct some very powerful and entertaining games with very little effort. Also, we can spice up our apps by making animated splash screens (a brief duration screen that pops up to introduce apps). In this chapter, we unleash our imaginations!

Prepare to have fun—before and after dragging blocks into some neat games.

In this chapter, we look at:

- Animation
- Splash Screens
- Bash the Alien! (Arcade-type games)
- Knowledge games
- GameClient Component

Let's get things in motion—and I mean that literally—by looking at animation in App Inventor.

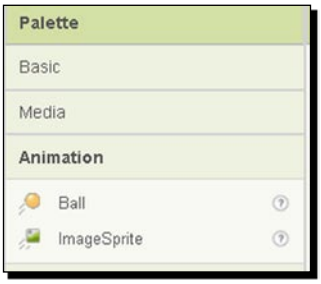
Animation

Animation is displaying a sequence of two- or three-dimension art rapidly so that it appears to move. We've all watched cartoons; we know what that means.

What makes animation work is showing the sequence fast enough so that an optical illusion called *persistence of vision* occurs, and we see it as smooth motion. I love doing animation!

In App Inventor, we have two animation components: **Ball** and **ImageSprite**. We find them in the **Animation** drawer in the **Palette** column of **Design** (see next screenshot). These components work on the **Canvas** component (found in the **Basic** drawer).

Over time (a sequence), we can control their movement, position, color, image, what they bounce off, and more. Up until the most recent series of updates (see *Changelog* and *Son of Changelog* sections in the previous chapter), movement was restricted to two dimensions (up and down, left and right). Now, we also have `z` (backwards and forward) and can move back and forth in layers! Again, as I mentioned in the last chapter, not true 3D but it is still nice to have.



"Wait, only *two* animation components?", you ask.

Yep, but like most App Inventor components, these are deceptively simple. We'll explore several uses for them in this chapter and show you a taste of their power.

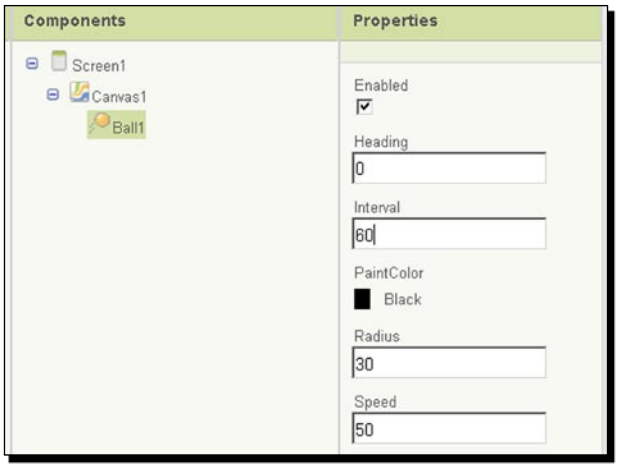
Why wait? Let's animate!

Time for action – bouncing a ball

This will bounce a ball off the walls inside our phone. It's simple, but fun to watch.
Here we go:

1. Drop a **Canvas** on the virtual phone screen in **Design**. Make its width Fill Parent, but do not worry about height (we'll fix that in **Blocks Editor**).
2. Drop a Ball onto the **Canvas**.
3. In the **Properties** column with the Ball selected (as in the next screenshot), change the **Interval** parameter to `60` (the default of `1000` is too jerky, `60` is much smoother). **Interval** is the time, in milliseconds, between movements.

Download from Wow! eBook <www.wowebook.com>



4. Set the Radius (size) to `30` and the speed to `50`.

That's it for the design of our bouncy ball. Don't worry that the ball rolled over to one side and hung up—we still have to build our blocks for proper operation.

After we get the ball bouncing right, come back and play with the previous settings and observe the actions of the ball. That's the fastest way of learning how this basic animation works.

Now we go to the **Blocks Editor** and pull together blocks so that the ball bounces around better.

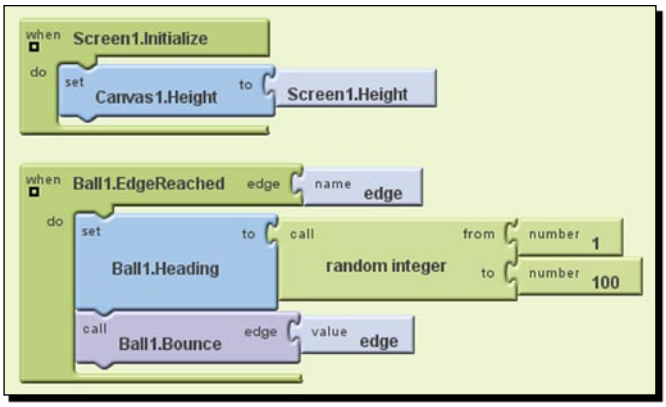
First (refer to the following illustration, please), we drag out a screen initialization block to fix our canvas height problem. I found that making the canvas Fill Parent in height in **Design** did not work (note: okay, later I found it *does* work, but this is still a good technique to know).

The simple fix is to use the **Canvas1.Height** button (from the canvas drawer), and plug a **Screen1.Height** block into it from the screen's drawer. This tells the canvas its height is the same as your device's screen, whatever that might be.

Next, pull out (from the ball's drawer naturally) the **Ball1.EdgeReached** frame, which automatically creates a name edge local variable (that is, one that works just within the framework as opposed to globally or anywhere).

We place a **Ball1.Heading** block and randomly change the heading (angle at which the ball bounces). We do this because otherwise the ball just bounces back and forth in a pretty uninteresting manner. Now, it will go all over the place.

Finally, we add a **Ball1.Bounce** block and get the value edge block from the **My Blocks**/**My Definitions** drawer, thus instructing the ball to bounce whenever it hits the edge of the screen—top, bottom, left, or right.


when Screen1.Initialize
do
set Canvas1.Height to Screen1.Height
when Ball1.EdgeReached edge name edge
do
set Ball1.Heading to call random integer from number 1 to number 100
call Ball1.Bounce edge value edge

What just happened?

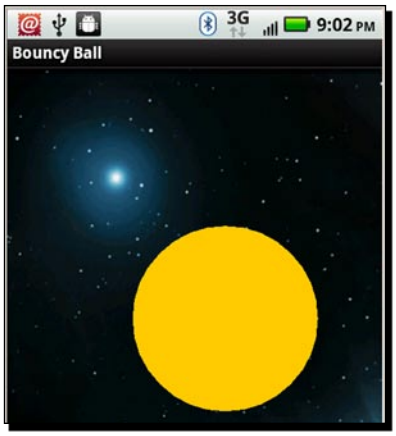
Watch it on your test device's screen. That bad boy is trying to bounce its way out of there!

Neat, but let's up the ante and give our ball the ability to move back and forth. We do this by randomly changing the size of the ball so that it apparently moves back and forth, thus giving an approximation of true 3D animation.

We use the **Ball1.Radius** block and feed it random numbers as shown in the next screenshot. We drop this block into **Ball1.EdgeReached** block. Watch that on your phone; now the ball is not only all over the phone and bouncing off the walls, but appears to be hitting the screen. Hope he doesn't break it, eh? I am not responsible if that happens.

Okay, but a black ball on a white background is not too realistic. I've uploaded a starry space background to Design, and I'll set that as the background image for the canvas and change the color of the ball to orange.

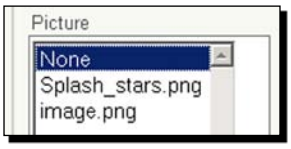
Try this with any photo or other graphic you might have on your computer. A `.jpg` or `.png` works best as a background. And, making it the size of your screen device will look a lot better as well. Here's how mine looks on my faithful Droid 2 sidekick:



Well, the background looks realistic, but not that ball. *Hmmm*, what was that other animation component we discussed? Right! The **ImageSprite** component; think I will try that and see what it does (playing in this manner is a great learning process, by the way, and fun to boot). You can adjust size in Design but uploading an object around 100 by 100 pixels will work well here.

So, we go back to Design and drop in an **ImageSprite** onto the canvas from the **Animation** drawer. We also upload an image—preferably something in a `.png` file with a transparent background. I'll be using my little alien critter you've seen several times already in this book.

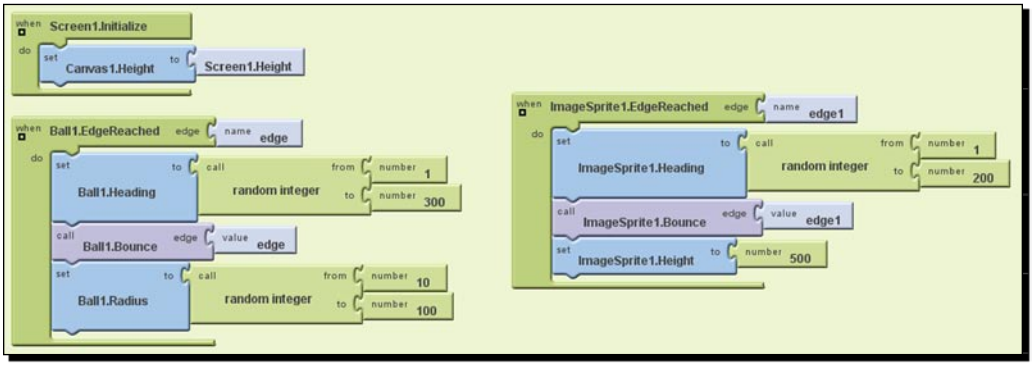
Attach your uploaded graphic to the image sprite by clicking in the **Picture** field in **Properties** column in **Design** and clicking on the graphic's name, as shown in the following screenshot:



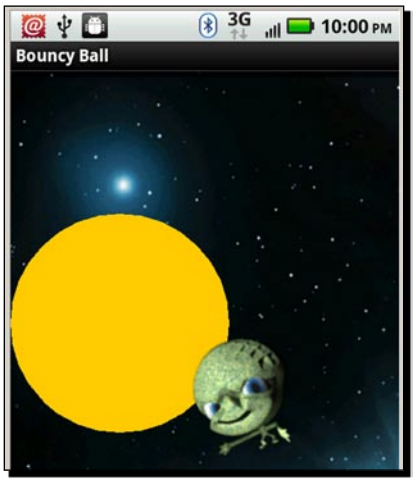
Set the speed of your sprite to `50` and the interval to `200`. These just being what achieve the effect that I want. You can always come back later and change them, or use various blocks in the sprite's drawer in **Blocks Editor** to make adjustments on the fly in your app.

Now, back to **Blocks Editor** and our still bouncing ball (assuming your test device is still running).

We just sort of mirror the ball, pulling out equivalent blocks from the **ImageSprite1** drawer.


when Screen1.Initialize
do set Canvas1.Height to Screen1.Height
when Ball1.EdgeReached edge name edge
do set Ball1.Heading to call random integer from number 1 to number 300
call Ball1.Bounce edge value edge
set Ball1.Radius to call random integer from number 10 to number 100
when ImageSprite1.EdgeReached edge name edge1
do set ImageSprite1.Heading to call random integer from number 1 to number 200
call ImageSprite1.Bounce edge value edge1
set ImageSprite1.Height to number 500

There! Looking at my result on my phone's screen (wish I could show you guys video), I got a great little animation. One which proves the old adage: "never play spaceball with an alien, *especially* if money is involved."



Here's another animation technique for balls and sprites, or for any mixture of those—*collisions* or bouncing not only off the walls (edges of the screen) but also each other.

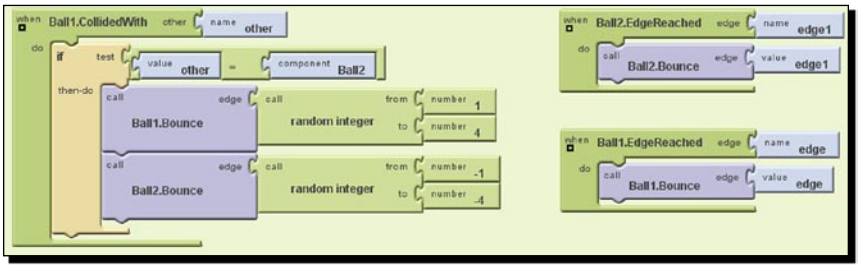
Let's do that; it's way cool.

Time for action – banging things off each other

1. In **Design**, start a new app design named *collision*, or whatever.
2. Drop in a **Canvas** component. Make it Fill Parent / Fill Parent. Upload and attach a background photo to it if you want.
3. Drop in a couple of **Ball** components onto the canvas from the **Animation** Drawer. Set the **Radius** (size) for each to `50`. Select different colors for each by clicking on the box under **PaintColor**.
4. In the **Properties** column for each, change **Heading**, **Interval**, and **Speed**. Perhaps `225` / `60` / `60` (the direction the ball first moves to start it bouncing, interval speed) for the first and `25` / `50` / `50` for the second—that looks pretty good to me, but I encourage you to experiment and get a feel for how these parameters affect movement on the phone's screen by changing interval and speed. We want balls moving at slightly different speeds and intervals to enhance a look of randomness.

   In the **Blocks Editor**:

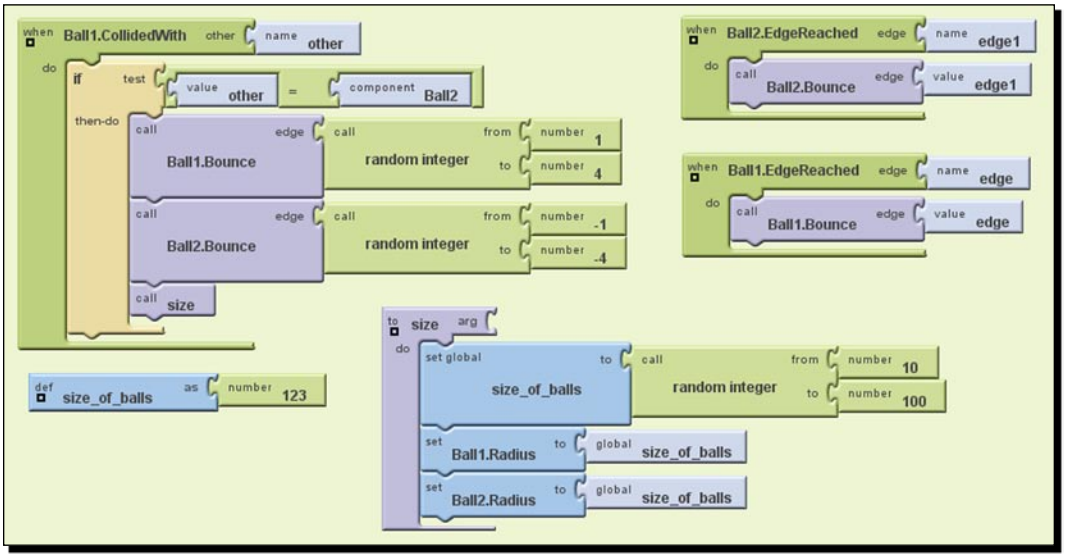
5. We only need one ball to respond if it collides with another (see next screenshot), so we pull out the **Ball1.CollidedWith** frame from the...? Right! **My Blocks**/**Ball1** drawer.
6. Populate the previous frame with blocks as shown in the next screenshot. In this frame, we:
   a. Use an **if then-do** block to test for when **Ball2** (other) touches **Ball1**.
   b. Click in **Ball1.Bounce** and **Ball2.Bounce** blocks so that the balls know what to do when they collide. For **Ball1**, we add a random integer from 1 to 4 (corresponding with North (`0` degrees, value `1`), Northeast (`45` degrees, value `2`), East (`90` degrees, value `3`), Southeast (`135` degrees, value `4`). For **Ball2**, we give it an opposite direction by using a random integer between `-1` and `-4`. South (`0` degrees, value `-1`), Southwest (`225` degrees, value `-2`), West (`270` degrees, value `-3`), Northwest (`315` degrees, value `-4`).
7. Finally, we add the two frames so that our balls know how to bounce off the walls.


when Ball1.CollidedWith other name other
do
if test value other = component Ball2
then-do
call Ball1.Bounce edge call random integer from number 1 to number 4
call Ball2.Bounce edge call random integer from number -1 to number -4
when Ball2.EdgeReached edge name edge1
do call Ball2.Bounce edge value edge1
when Ball1.EdgeReached edge name edge
do call Ball1.Bounce edge value edge

What just happened?

Watch the previous on the screen of your test device. Looks like an exciting professional wrestling match, huh?

Now, let's modify it to give the illusion of three dimensions by changing the size of the balls. We want them to seem to stay together for this effect so (see next screenshot) we add a procedure (**size**) that changes both balls according to a single random integer between `10` and `100`. We also declare a global variable, **size_of_balls**, to hold that value until it gets applied to both balls.


when Ball1.CollidedWith other name other
do
if test value other = component Ball2
then-do
call Ball1.Bounce edge call random integer from number 1 to number 4
call Ball2.Bounce edge call random integer from number -1 to number -4
call size
def size_of_balls as number 123
to size arg
do
set global size_of_balls to call random integer from number 10 to number 100
set Ball1.Radius to global size_of_balls
set Ball2.Radius to global size_of_balls
when Ball2.EdgeReached edge name edge1
do
call Ball2.Bounce edge value edge1
when Ball1.EdgeReached edge name edge
do
call Ball1.Bounce edge value edge

We have a good idea by this point how to animate with balls and sprites, but there's more!

Have a go hero – playing with the other animation blocks

Take some time here, and enjoy yourself in adding other features to the balls. We'll use the previous example as a starting point.

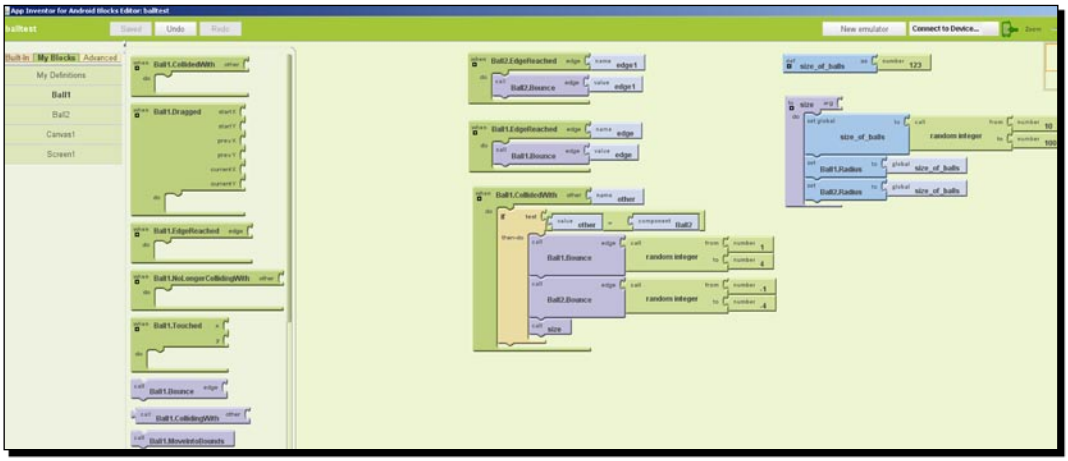
Click on, for example, the **Ball1** drawer. As we know, this opens up a long list of the blocks automatically created when we first dropped that component onto the virtual phone screen in **Design**.

These blocks (the same also with minor differences for sprites) give us a lot of ways to set up *user interaction*.

User interaction—such as the ball or sprite reacting when touched or the user having the ability to drag balls to specific locations on the screen—makes creating games possible.

The following is the **Blocks Editor** screen for our current example with the **Ball1** drawer open.

When a drawer is open, placing the mouse cursor on top of a block brings up a short explanation message about what the block does.



Oh, and you might want to slow the balls down some for this or you won't be able to, for example, touch them in a game.

Pop over to **Design** and change the speed to `10` for both balls, which slows them down nicely. It's always good to develop and test your games in slow motion.

We now go back to our buddy, the **Blocks Editor**, so that we can add some interaction. Because, right now, the balls just bounce around and hit each other, and we have no way to affect the outcome—which would be a very boring game.

What to do?

Ummm. If we could drag one of the balls around so that the other ball *could not* hit it, we'd have the basis for an interesting game.

In this game (feel free to create and publish it), you get a point every time the balls collide. The object of the game is to get the *least* number of points possible in a given time period.

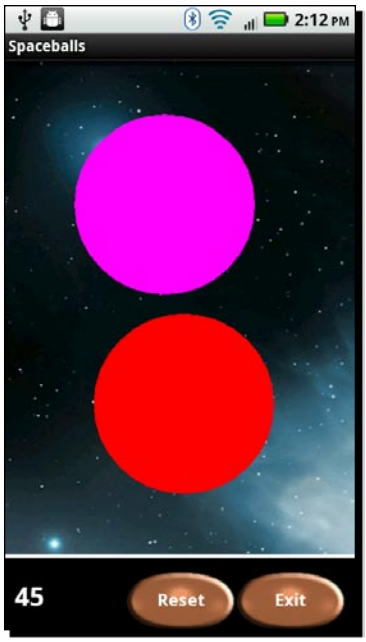
Right, we skip our way back over to **Design**. We don't need much more to turn this into a full-fledged game; just a label to display the score, a **Reset**, and an **Exit** button.

To make room for those, just change the height of the canvas to, say, `450` or `compute dynamically using Screen1.Height – 100 (or some other value leaving enough space for our label)` (you can always play with settings until things have the look you want).

Put a horizontal arrangement (our most trustworthy formatter) in the space at the bottom and do Fill Parent for its width. Drag a label and two buttons in place.

For canvas height, you can set size to **Screen1.Height–HorrizontalArrangement1.Height**.

The complete screen for our game (I'm calling mine *SpaceBall*) now looks something probably (depending on your own design and background) like the following:

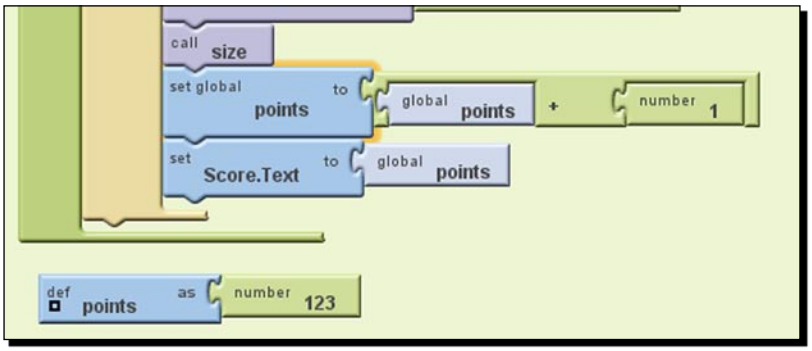


In the **Blocks Editor** we now need to make our new features work.

First, let's keep score. We need to create a **points** global variable (declaring it as a number variable; see next screenshot).

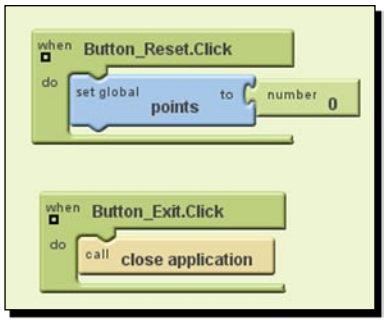
The place to increment and display our score is in the **Ball1.CollidedWith** frame. And all this takes is adding the **points** global variable block beneath the call to the **size** procedure and adding `1` to it (which happens every time a collision occurs).

Then, we put the **points** value into the **Score.Text**, and it gets displayed on the phone's screen.



Setting up the two buttons is equally easy.

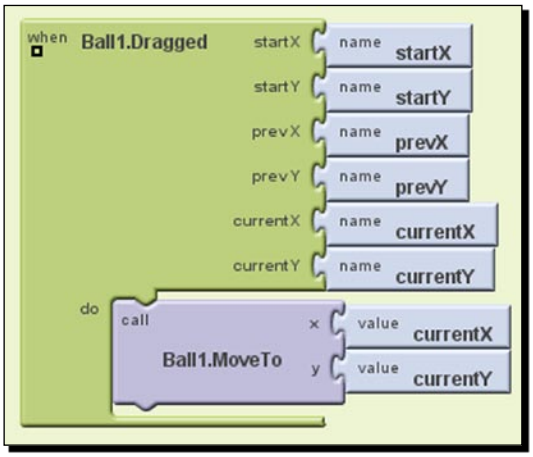
The **Reset** button (see next screenshot), when tapped, sets the score back to zero (you'll also want a button that sets the score back to zero). The **Exit** button does our standard favor to the user of properly closing the app.



Now, we add the interaction part—got to give the game player something to do!

From the **Ball1** drawer (**Ball1** is our hero, **Ball2** the spawn of evil attempting to bump the score up to infinity), get the **Ball1.Dragged** frame.

This following frame looks complex at first, but it's not really. All those local variables are automatically created and added to the **My Definitions** drawer, but all we need are **currentX** and **currentY**. `current` is used to get coordinates of your touch, `prev` is used to get the last previous coordinates, and `start` is used to get the initial coordinates of the image. Click those into a **Ball1.MoveTo** block, and our game is ready to play (increase the speed of the balls to `20` as a start).



What just happened?

We created two balls whizzing back and forth and, by the simple addition of the **Ball1.Dragged** block, turned it into a game. Grabbing the hero ball by touching and trying to move it out of the way of the "evil" ball is appropriately mushy and hard to do. A fun game created with little effort.

Have a go hero – add features to the game

Some of the other things you'll want to do before publishing this might include:

1. Set up a **Clock** component time to limit duration of the game.
2. Add levels with complications such as increasing the speed of the balls.
3. Include a list to keep track of high scores and **TinyDB** to make it persistent.
4. Store high scores into `TinyWebDB` so that users can compete against each other and see others' high scores.
5. Add a sound when the balls collide.
6. Do something to show when **Ball1** is under the player's control.

Lots more things you can add, but this is how a simple arcade-type game is developed in AI.

Some more basics of App Inventor animation

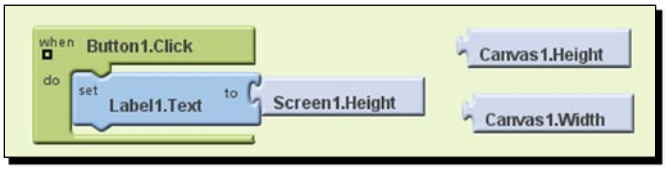
A few more things before we continue.

We can move balls and sprites using (as in the case of a ball) a **Ball1.MoveTo** block as shown in the following screenshot. Just plug in numeric variables with the `x` and `y` values.

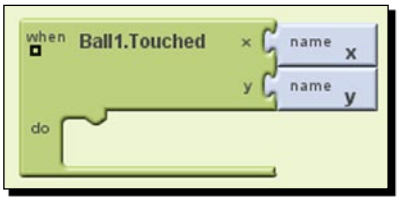
How do you find the `x` and `y` coordinates, and what are they?

That would be the width (`x`) and height (`y`) of your canvas. If the canvas is `320` by `400`, the center would be `160` for the `x` value and `200` for the `y` value.

If you're unsure, a simple button/label test lets you find these values by plugging in the width and height blocks. This works for the screen as well as for a canvas or any other component with height and width blocks.

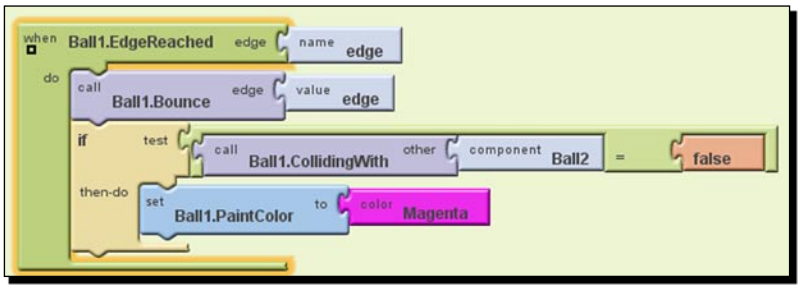


Ball1.Touched (or **Sprite1.Touched**) is another method of user interaction. Whatever you want to happen when that object gets touched is put within the frame. The local variables are the current `x` and `y` position when the touch occurs. The value blocks for those appear in the **My Definitions** drawer automatically.



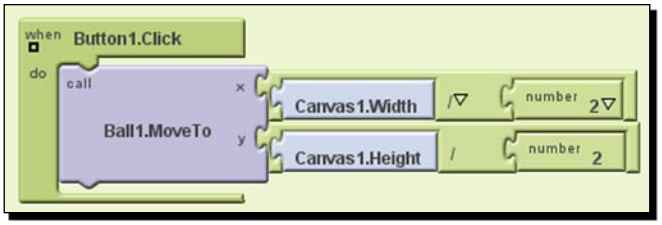
The **Ball1.CollidingWith** block gave me an idea for embellishing *SpaceBall*. We can add a test in the block frame that fires when the ball touches an edge to see if it is `false` (not touching that evil old **Ball2**), as shown in the next screenshot. This means we can have **Ball1** change back to its normal color.

Of course, we also change the **Ball1** paint color to yellow in the **Ball1.CollideWith** frame so we have a color to change from. Works nicely.



Centering an object (ball or sprite) is a snap. Set `x` to the width of the canvas divided by `2` and `y` to the height divided by `2`.

To make centering even more accurate, try this out. Change it so that you calculate (**Canvas1.Width / 2**) - (**Ball1.Width / 2**). And do the same thing for height. This will account for the size of the ball and give you a true center.



Or for these points:

- Top Left Corner: 0, 0
- Top Center: width/2, 0
- Top Right Corner: width, 0
- Left Center: 0, height/2
- Right Center: width, height/2
- Bottom Left Corner: 0, height
- Bottom Center: height, width/2
- Bottom Right Corner: height, width

Or anywhere else in there you can do the math for, eh? Stay tuned, there's more.

Download from Wow! eBook <www.wowebook.com>

Now, time to look at layers, and one great application for layers is in making splash screens (the *splash screen* is a limited-duration screen that introduces an app like the beginning credits screen of a movie or TV show).

Splash screens

Let's go back to *SpaceBall* and add a splash screen.

One of the most recent features added to App Inventor, *Z-layering*, makes this easier and more powerful.

As we've discussed already, 3D animation uses the conventions of x-axis for side to side, y-axis for up and down, and z-axis for forward (toward us) and backward (away from us).

AI does not have motion in the z-axis yet, but it *does* now have layers. Putting balls or spites in different layers means they can pass each other without colliding. One object will appear to pass in front of or behind another depending on which layer they're in.

The higher the number parameter of a layer block is, the closer it is to us. We can set this parameter either in the **Properties** column of **Design** (a **Z** field has been added to the **X** and **Y** there in previous builds of App Inventor), or on the fly in the **Z** blocks automatically created in each ball's or sprite's drawer in the **My Blocks** tab of **Blocks** Editor.

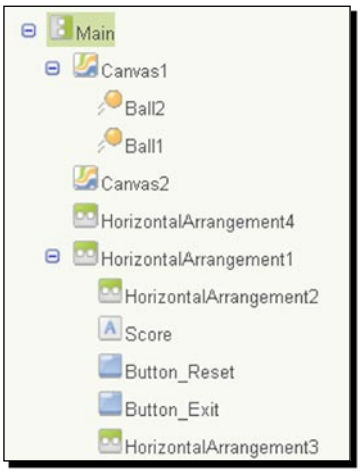
The following is an example of putting two image sprites in different layers. **ImageSprite1** will pass in front of **ImageSprite2**.

This may seem a minor enhancement, but it opens a lot of possibilities, not least of which is animated splash screens. We can now have text passing in front of objects and all sorts of good stuff. My imagination was sparked, and I hope yours will be after we create the following splash screen example.

Time for action – designing a splash screen

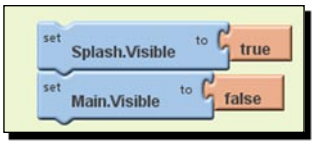
Whether you have a static splash screen (nothing moving on it) or an animated one, you'll need multiple screens in your app. So, we first convert *SpaceBall* to more than one screen.

The best way to do this is to put everything we have now into a vertical arrangement (make it Fill Parent/Fill Parent). Rename the vertical arrange to **Main** (or **Home** or whatever you like for your first page name). Pull everything we now have into it so that it now looks like the following:



It's just easier—based on my experience—to use vertical arrangements for virtual "screens" or "pages." (These are not physical discrete screens like AI will get sooner or later, but for now they do the job). We can then primarily use horizontal arrangements for formatting on each "screen". Again, merely something I've found simplifies designing apps.

We navigate between screens by making one visible and the other not visible.



Now, let's design our splash screen.

First, collapse **Main** in the **Components** column, and uncheck **Visible** in the **Properties** column, avoiding confusion as we create the splash screen.

Add a vertical arrangement, making it Fill Parent/Fill Parent (remember, we do this to cause it to totally fill the phone or other device's screen). Rename it to **Splash**. This will be our splash page (or screen. I tend to use those terms interchangeably).

We need something to do our animation on. In AI, that would be a **Canvas** component, so add one, placing it inside the **Splash** vertical arrangement and also making it Fill Parent/Fill Parent.

Finish out the design by dropping three image sprites on the canvas (I'm envisioning three animated elements for my intro production, which will have music, too).

We also need a clock (so that the splash page only displays briefly). We also need to have a player for the music. **Clock** is in the **Basic** drawer; player's in the **Media** drawer. Just drop them on the canvas.

Okay, now we want four graphic elements—a background image for the canvas, and whatever you like for the three sprites. To speed up this learning process, you are welcome to use my art—this entire example app is available at: `http://arrsoft.com/examples/SpaceBall.zip`.

Here's the main logo (I created it in Photoshop):

Now, lastly before completing the design of the splash page, we must configure the three sprites.

The alien graphic (or whatever you're using) gets attached to **ImageSprite1**. In the **Properties** column with this sprite selected, we set the Heading to `30` (need an angle to get motion started and keep it from hanging up), Interval to `60` (nice and smooth motion), Speed to `35` (we want him to be zippy), and, as already mentioned, the **Z** layer to `2`.

After you get everything working, you can come back and play with the settings for the sprites.

Set the other two sprites' **Heading** to `30`, their Interval to `60`, and their speed to `8` (we want them slower than the alien). Attach (if using my art) `chills.png` in **ImageSprite2** and set its **Z** layer to `1`. Attach `thrills.png` to **ImageSprite3** and make its **Z** layer `3`.

Also, attach some music to the player, such as the royalty-free piece included in my ZIP file.

What just happened?

The design part is now complete.

In my setup, I have my little alien bouncing around on the splash screen in layer `2` with the words *Thrills* (layer `3`) and *Chills* (layer `1`, farthest back). So, he moves *between* those two words, avoiding collision since he's on a different layer.

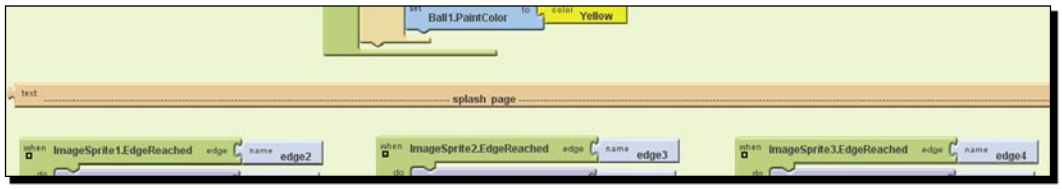
So, walk with me over to **Blocks Editor**, and let's make all that actually work.

Time for action – making a splash with the splash page

Before we do what little is needed in bringing the splash page to life, here's a tip you might find useful.

As apps become more complex, the **Blocks Editor** gets cluttered with block groups scattered here and there. App Inventor doesn't care where they are; it knows what sequence to run them in. And here we go, adding more groups for our adventure in splashing.

What I like to do is grab a **text** block from the **Built-In**/**Text** drawer, fill it with a dashed line (see next screenshot), and add the name of the page. It makes a nice divider bar and relieves the visual confusion. Since it is not plugged in to anything, it does not affect operation of the app.



But, to work—we have three minor tasks to accomplish:

1. Start the sprites moving.
2. Set up the initialization of the app to include starting the music player, making the **Splash** page visible, and the **Main** page not visible yet.
3. And, determining how long the Splash page displays.

First, we put our three sprites into motion by instructing them to bounce off the walls (edges of the screen) as shown in the next screenshot.

Remember, App Inventor has already created the edge local value blocks you need to click into the bounce blocks—get them from the **My Definitions** drawer.

when ImageSprite1.EdgeReached edge name edge2
do call ImageSprite1.Bounce edge value edge2
when ImageSprite2.EdgeReached edge name edge3
do call ImageSprite2.Bounce edge value edge3
when ImageSprite3.EdgeReached edge name edge4
do call ImageSprite3.Bounce edge value edge4

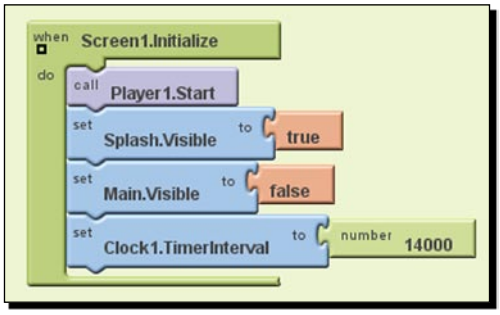
If you are watching your test device (as you should be both during the designing and block-building phases of creating apps), you will see our three sprites moving right along. Good so far.

The **Screen1.Initialize** block (what happens when the app starts) has blocks that kick off the music, make the **Splash** page visible and hide the **Main** page, and set up the duration that our splash screen is visible. And all that is shown in the next illustration.

The **Clock1.TimerInterval** block is in the **My Blocks/Clock1** drawer. Plug a numeric variable block into it and set it for the duration time.

The clock uses milliseconds (1,000 milliseconds = one second). So, `14,000` is about 14 seconds, which causes the screens to switch over just as my music ends.

You get this timing exact by experiment and watching on your test device. Took me about three tries to get the splash screen ending just as the music did, but that's good production value and people notice things like that.

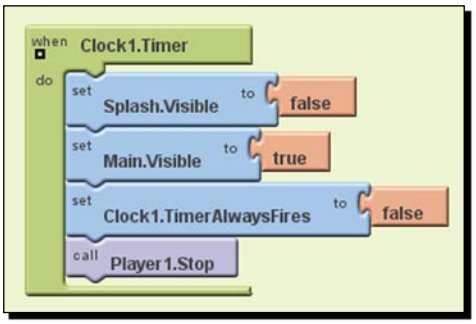


Okay, in the last block in **Screen1.Intialize** earlier, we set the timer interval to `14` seconds. So, now it is time to tell the clock what happens when those 14 seconds are up. To do that, pull a **Clock1.Timer** frame out of the **Clock1** drawer.

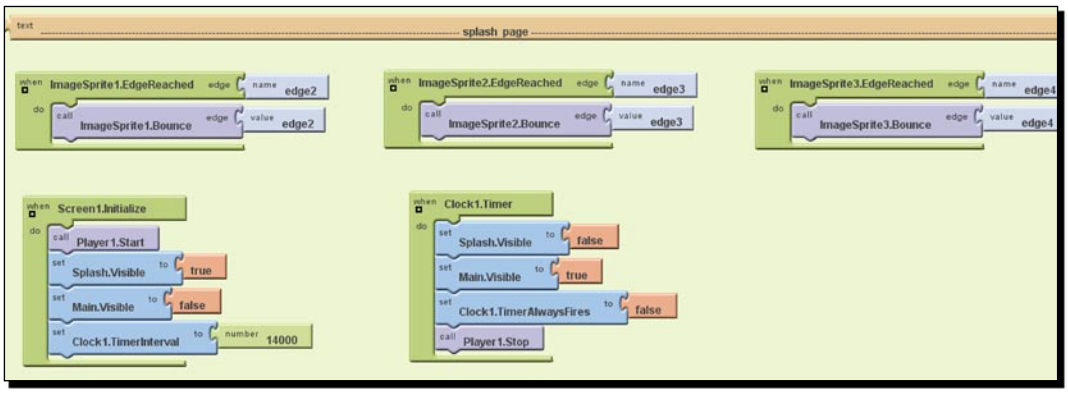
In the frame—and, remember, this happens after the delay time is up—we hide **Splash** and make **Main** visible so folks can now play the game, awed as they are by our masterful intro.

We also tell the clock we're done with the delay by setting **Clock1.TimerAlwaysFires** to **false**.

If, for instance, we had an event we wanted to occur every five minutes, we could set the clock timer interval for that, and by not turning off the clock as in the next screenshot, that event (the blocks in this frame) would fire off every five minutes so long as the app is running.



That does it: three simple steps and our splash page now works. Here are the complete blocks for this segment of our game (that is, the ones running the splash screen):



What just happened?

We now have a snazzy opening to **Spaceball** (see the next screenshot). Yes, the game itself still needs a lot of work, but it was my purpose here to give you guys a big start on methods of developing games in App Inventor, not to do the whole thing, eh?

What I would like to see you take away from this example is the sense of wonder I enjoy in seeing how easy it is to use Google App Inventor. What you might at first think really complicated—such as constructing a splash page—turns out just to be pulling together a few blocks.

But, so much for civility, let's bash some stuff!

Bash the Alien!

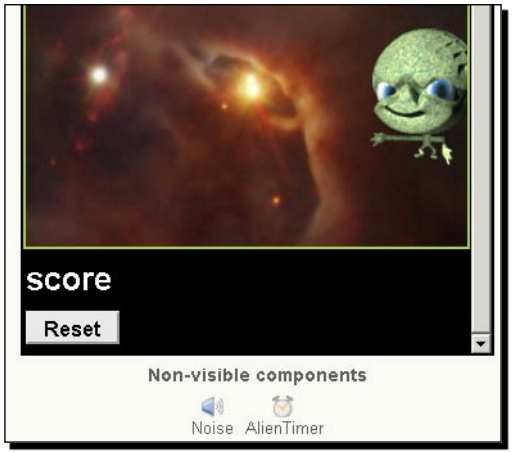
In the previous game, my little alien just had a walk-on part (okay, a *spin-on* part) in the splash animation. Our hero ball and that mean old evil ball were the stars. The following game gives the leading role to the alien, although he does take a bit of a beating for it.

Bash the Alien! is the first game app I wrote in App Inventor. In fact, it's about everyone's first game because it is based on *MoleMash* on the AI tutorial site, which in turn came from an old arcade game called *Whac-a-Mole*™. We all just like bashing stuff in games, I suppose.

I elaborated it a bit (see the next screenshot). So, let's discuss it. Although the techniques are somewhat the same, the game itself is played totally differently. It also makes use of the clock timer in the game and hides the alien from time to time, and other little tricks to make it harder than it looks to play—which is precisely what a good game should do.

The design part of this type of game is minimal. We need a canvas (I named mine **Space**), setting it to Fill Parent wide by `420` high leaves room on my Droid 2 for a **score** label and a **Reset** button.

Drop in a sound and a clock non-visible components—I've renamed mine here:



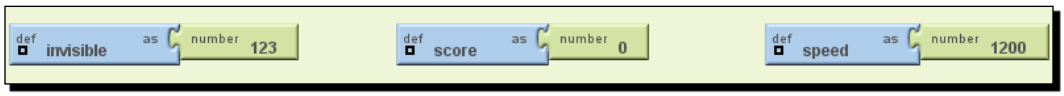
Now, we dance over to the **Blocks Editor**.

To make it easier to follow along, the source is at `http://arrsoft.com/examples/AlienBash.zip`.

Time for action – throwing blocks to Bash the Alien!

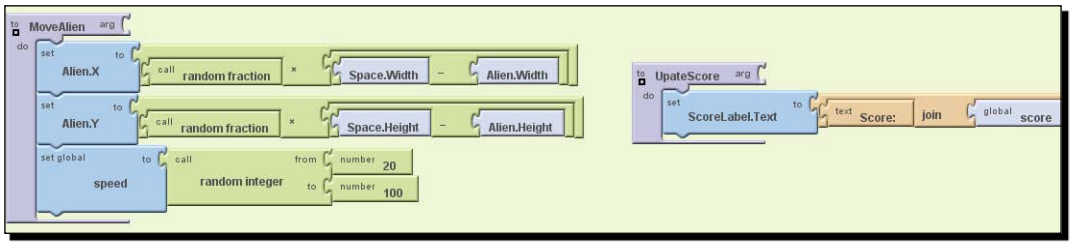
Along the top of the Blocks Editor main window, we define three global numeric variables. Although each has a different value, that really does not matter—they get new values as the app runs.

The three, in order, control invisibility (a nasty little trick the alien has of ducking into hyperspace and emerging at a random location), the score (how many or how few times you manage to tag him), and speed (he changes speed and direction randomly).



Next, we add two procedures:

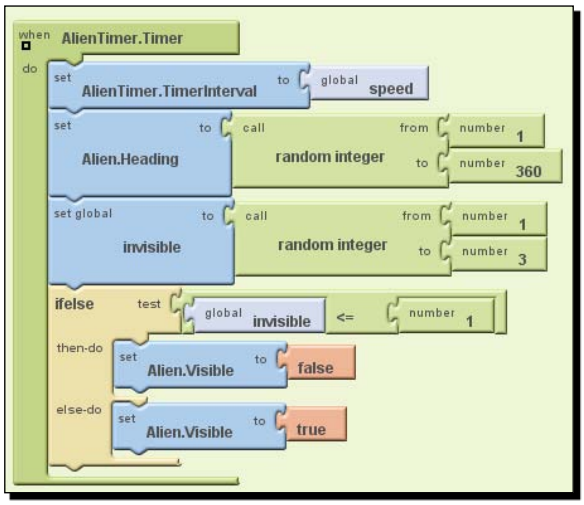
1. **MoveAlien** does the random stuff. The `x` and `y` coordinates of the alien's position change randomly and he zips there at varying speeds.
2. **UpdateScore** keeps track of how many times he gets touched (darn few, he's a whirling dervish and delights in making us all look like slow, clumsy humans—oh, wait, we are).


to MoveAlien arg
do
set Alien.X to call random fraction × Space.Width − Alien.Width
set Alien.Y to call random fraction × Space.Height − Alien.Height
set global speed to call random integer from number 20 to number 100
to UpateScore arg
do
set ScoreLabel.Text to text Score: join global score

And, of course, we put in an edge event (see the next screenshot) so that he's bouncing off the walls and changing directions that way, as well.

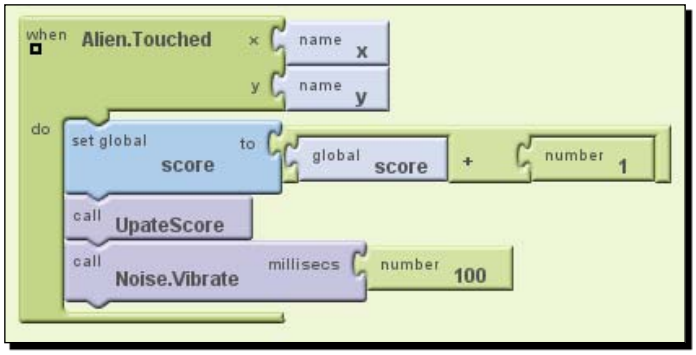
We use the clock timer to let him do yet more nastiness. First, we change his speed, based on the random speed calculated in the **MoveAlien** procedure or, to be more specific, the amount of time lapsing before he pops off in a new direction.

The direction (heading) he takes off in is also randomized. And that mean little alien (oh, he's a crafty one, he is) goes briefly invisible one out of three times.

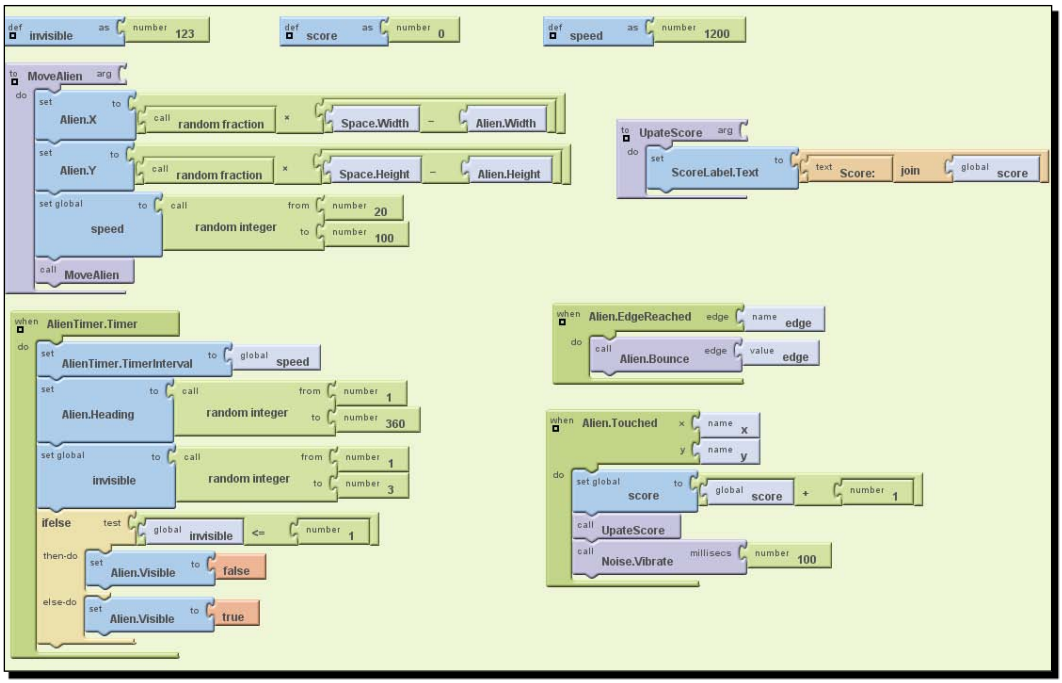

when AlienTimer.Timer
do
set AlienTimer.TimerInterval to global speed
set Alien.Heading to call random integer from number 1 to number 360
set global invisible to call random integer from number 1 to number 3
ifelse test global invisible <= number 1
then-do set Alien.Visible to false
else-do set Alien.Visible to true

Finally, the few times he gets touched, here's what happens (in the next frame).

The score gets incremented by one, and the phone vibrates, letting the player know a touch happened. The **UpdateScore** procedure is called, which displays the new score on the phone's screen.


when Alien.Touched x name x y name y
do
set global score to global score + number 1
call UpateScore
call Noise.Vibrate millisecs number 100

Put them all together and it looks like the following screenshot, which might have made some of us nervous (like me) if we had come on it all at one time. However, I think that time is passing as we now see how the block groups fit together and more easily come up with the block groups that perform what we had imagined for the app from first concept.


def invisible as number 123
def score as number 0
def speed as number 1200
to MoveAlien arg
do
set Alien.X to call random fraction × Space.Width − Alien.Width
set Alien.Y to call random fraction × Space.Height − Alien.Height
set global speed to call random integer from number 20 to number 100
call MoveAlien
to UpateScore arg
do
set ScoreLabel.Text to text Score: join global score
when AlienTimer.Timer
do
set AlienTimer.TimerInterval to global speed
set Alien.Heading to call random integer from number 1 to number 360
set global invisible to call random integer from number 1 to number 3
ifelse test global invisible <= number 1
then-do set Alien.Visible to false
else-do set Alien.Visible to true
when Alien.EdgeReached edge name edge
do call Alien.Bounce edge value edge
when Alien.Touched × name x y name y
do set global score to global score + number 1
call UpateScore
call Noise.Vibrate millisecs number 100

What just happened?

Our alien dares us to touch him and our own app makes that truly difficult. Fun, huh?

And, while arcade-style games exercise our reaction speeds, knowledge games give our brains a workout. Let's build one of those next.

Knowledge games

Google emphasizes using App Inventor in education. That certainly makes a great deal of sense as the ease of creating apps in AI's block-building environment makes learning the concepts not only easy but quite enjoyable.

Knowledge games are apps that teach. I've got an extensive example next—**Mr. Wise Owl's Quiz Game**—which should appeal to both students and teachers. I've a fond spot for teachers—my mother taught for over 40 years and I, myself, went to school. Yes. I did. For a *long* time.

For ease of following along, get the source at: `http://arrsoft.com/examples/Mr_Owl.zip`.

As a reference, and to get your appetite whetted for this game and the concept behind it, all of which I'm giving you guys, here's what the screen looks like:

This app, by utilizing the power of the web, is actually a framework for learning. Teachers, by posting simple textfiles on a website, make new quizzes available for students to download and play in this app.

The example shown here is a State capital quiz, but this same app can load quizzes on a wide range of subjects. As you'll see in the next few pages, it's pretty complete already, although certainly embellishments are needed to really make it hot.

And, like most apps, we don't need a lot in **Design** to accomplish all this power. The **New** button downloads the next quiz. Later, we'll probably want to make it so that any number of quizzes can be in the game at any one time, using an additional screen to let the student decide which to load and play.

The States and their capitals textfile is on one of my sites and the URL configured in the source file. Please feel free to use it.

Next down, we have a **Question** button, which only has to be used once to start the game. From then on, as each question is answered, the score is shown and the next question pops up. We could, now that I think on it, probably do away with this button totally.

A vibration indicates a miss and a peppy little sound a correct answer, giving immediate reinforcement to the joy of learning.

The four choices for answers below that are also buttons. Tapping—in this case—on what the student believes to be the capital of the State in the question causes it to score a *yes* or *no* and move us on to the next question.

The **Reset** button lets the user zero out the scores and start afresh—after all, this is a self-learning, self-paced experience and one that we make as fun as possible.

The **Exit** button, as ever, lets us exit the app, taking it politely out of memory and not hogging resources when not in use. The quiz and scores are made persistent by **TinyDB**, so when the app is called again, we continue from where we left off.

TinyDB, two **Web** components, a **Notifier** and a **Sound** component, are the non-visible components we add to our design.

And, of course, I spent a lot of time on formatting. By putting everything in a vertical arrangement (entitled **Home**), it will be easy to add more virtual screens to the app.

Again, if you download the source, you get all my formatting to study in addition to all the blocks.

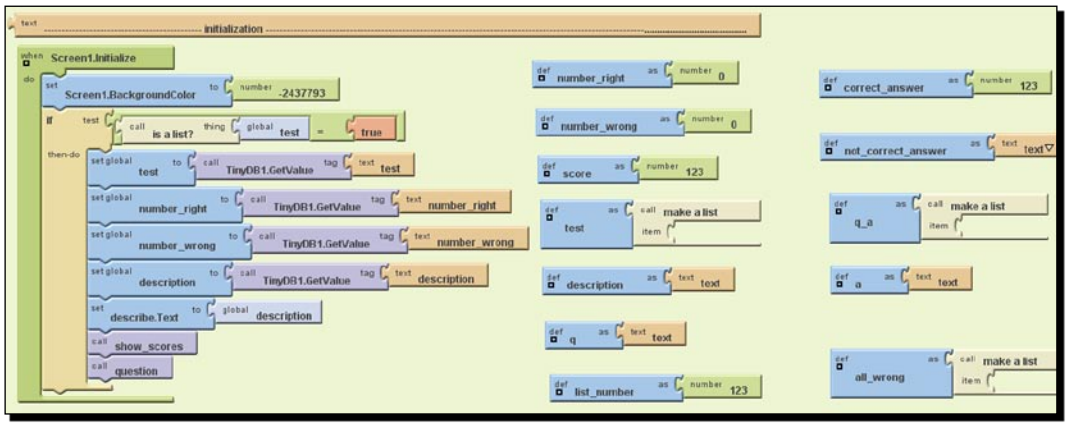
Now, if you will, hop with me over to the **Blocks Editor**. We'll look at the logic behind the scenes. There's a lot of stuff, so this is just an overview, but you'll find that dividing things into separate block groups shows you this is a pretty simple concept after all.

We begin at the beginning: what occurs when the app first initializes.

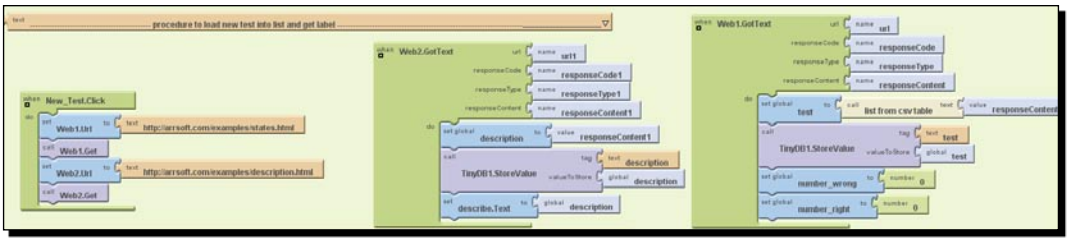
First, we need 12 global variables, declared as shown in the next screenshot. I'll tell you about what these are used for as we take an overview of each major block group.

Download from Wow! eBook <www.wowebook.com>

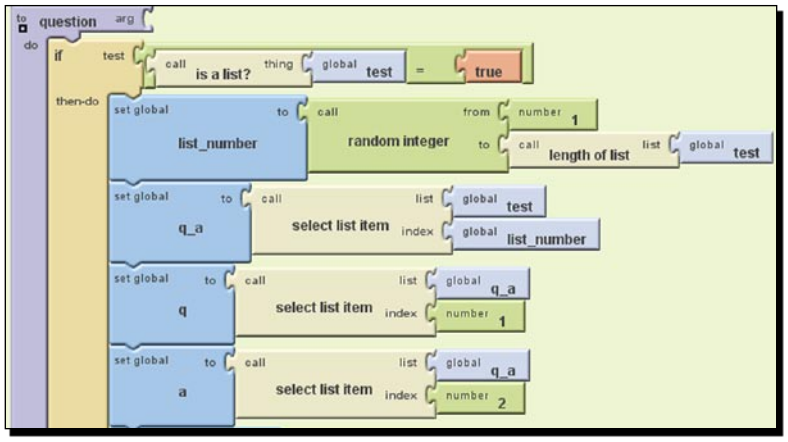
In the screen initialization frame, we set the color of the screen. Next (inside an **if** block as an error trap against empty strings) we recall the data from **TinyDB**, show the description of the game (downloaded from the web along with the questions and answers for each quiz), show the scores, and call a question so the student has something to start on.


initialization
when Screen1.Initialize
do
set Screen1.BackgroundColor to number -2437793
if test call is a list? thing global test = true
then-do
set global test to call TinyDB1.GetValue tag text test
set global number_right to call TinyDB1.GetValue tag text number_right
set global number_wrong to call TinyDB1.GetValue tag text number_wrong
set global description to call TinyDB1.GetValue tag text description
set describe.Text to global description
call show_scores
call question
def number_right as number 0
def number_wrong as number 0
def score as number 123
def test as call make a list item
def description as text text
def q as text text
def list_number as number 123
def correct_answer as number 123
def not_correct_answer as text text
def q_a as call make a list item
def a as text text
def all_wrong as call make a list item

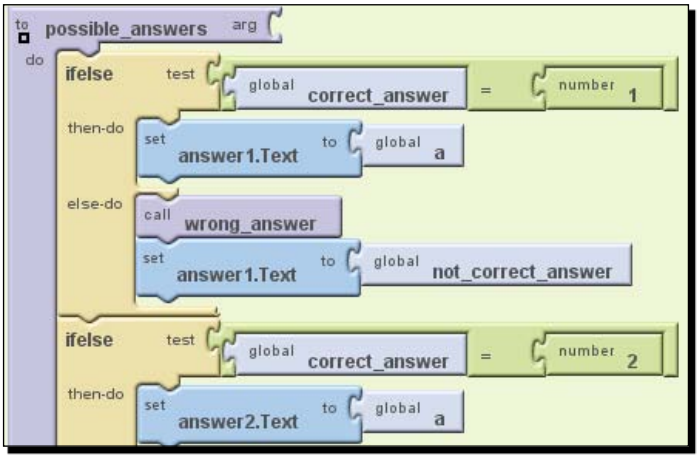
The next section is where the **New** button uses the two **Web** components to get the description and the questions and answers from the teacher's website. Once downloaded and the Q/A turned into a list, they are saved by **TinyDB**. As you elaborate this app, you'll want to add a third web component to get the question—that obviously needs changing for each quiz.



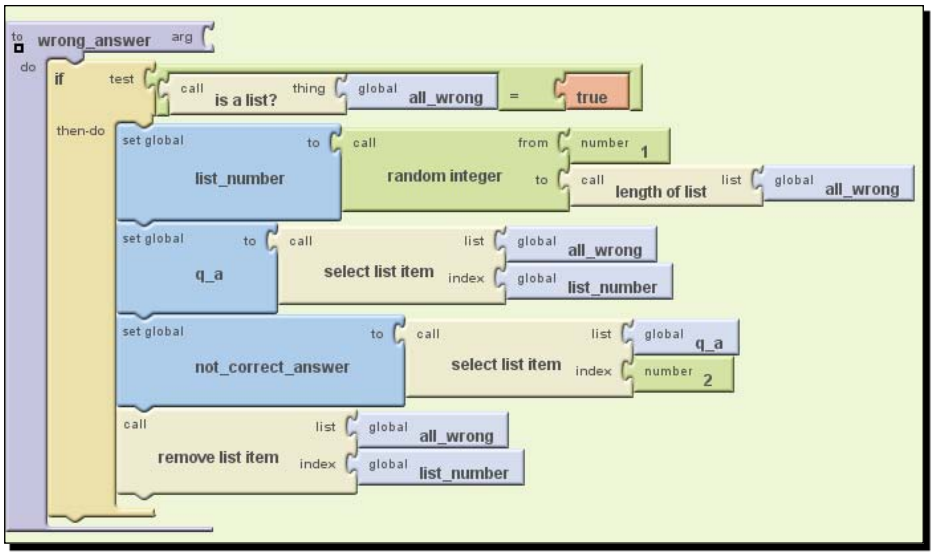
The **question** procedure (also in an error trapping **if** block) randomly selects a question/answer pair from the list, converting it out into two global variables: **q** and **a**. The question for the student is formatted and a list made of all the *wrong* answers. **Question** gets called by the **Question** button or when a question is answered or in initialization. Also, which of the four answer buttons gets the correct answer is randomly selected.


to question arg
do
if test call is a list? thing global test = true
then-do
set global list_number to call random integer from number 1 to call length of list list global test
set global q_a to call select list item list global test index global list_number
set global q to call select list item list global q_a index number 1
set global a to call select list item list global q_a index number 2

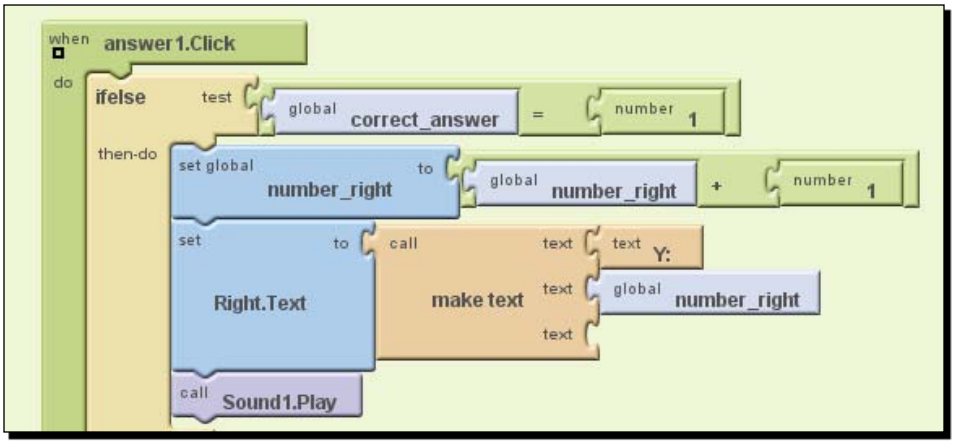
When **Question** finishes, it calls **possible_answers**. This procedure checks each of the four answer buttons. If the button has the correct answer, it displays it. If not, it calls the **wrong_answer** procedure, gets a wrong answer (a random selection of all the other State capitals), and displays that as the choice.


to possible_answers arg
do
ifelse test global correct_answer = number 1
then-do set answer1.Text to global a
else-do call wrong_answer
set answer1.Text to global not_correct_answer
ifelse test global correct_answer = number 2
then-do set answer2.Text to global a

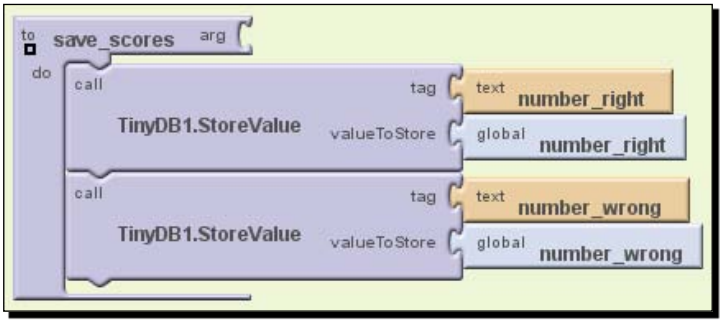
The **wrong_answer** procedure selects a wrong answer from the **all_wrong** list built-in **question** when the right answer was selected. It removes that answer from this temporary list so that duplicates do not occur.


to wrong_answer arg
do
if test call is a list? thing global all_wrong = true
then-do
set global list_number to call random integer from number 1 to call length of list list global all_wrong
set global q_a to call select list item list global all_wrong index global list_number
set global not_correct_answer to call select list item list global q_a index number 2
call remove list item list global all_wrong index global list_number

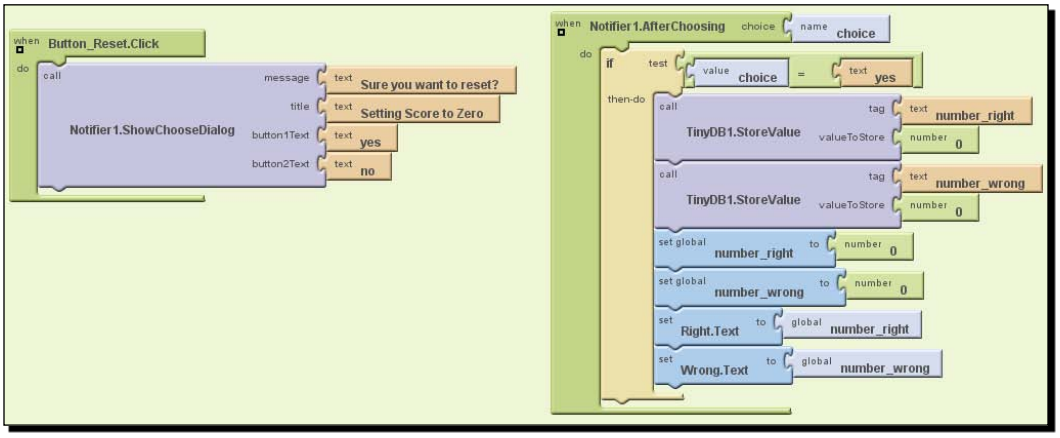
All four answer buttons are set up so they play the success sound or vibrate a fail and show the new score, and **question** calls the next question and **save_scores** records the results.


when answer1.Click
do
ifelse test global correct_answer = number 1
then-do
set global number_right to global number_right + number 1
set Right.Text to call make text text text Y: text global number_right text
call Sound1.Play

The **save_scores** procedure uses **TinyDB** to make the scores persistent.

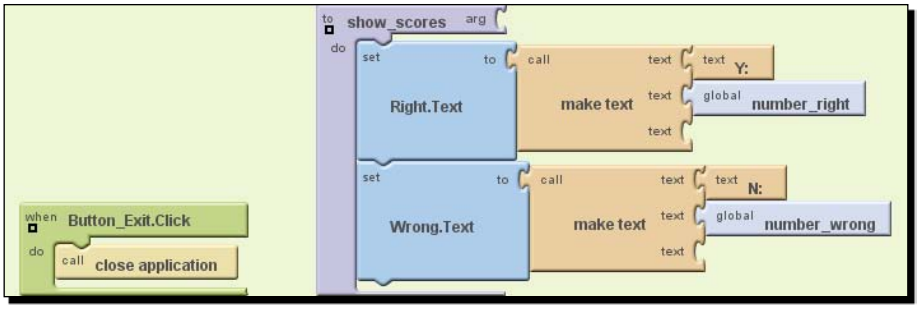


The **Reset** button does a bit more than one might expect for safety's sake. Basically, it resets the scores back to zero, but we use the **Notifier** component to make sure the user really *wants* to do this, and then to zero out the scores both in the global variables in **TinyDB**. The following condition does not check for a 'no' response because we simply want to close the notifier without updating any information.



By the way, the **New** button needs a **Notifier** like the previous technique as a protection from overwriting a quiz the student is still enjoying.

Finally, we have our **Exit** button and a procedure the initialization frame uses to show the scores when it first comes up. You could make a call to this procedure from the four answer buttons and condense those blocks. Naturally, I just thought of this now, but I'll leave that to you.


to show_scores arg
do
set Right.Text to call make text text Y: text global number_right text
set Wrong.Text to call make text text N: text global number_wrong text
when Button_Exit.Click
do call close application

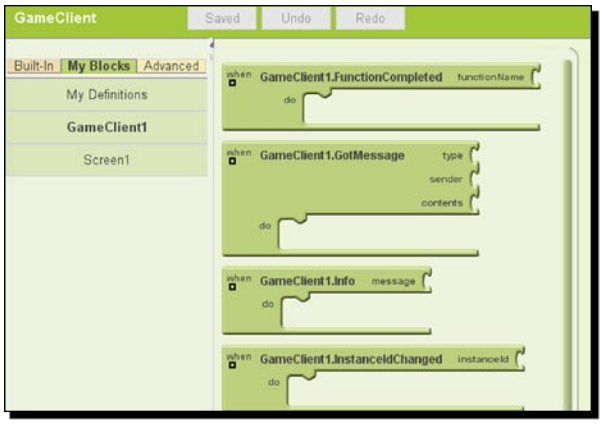
Okay, that's a good introduction to the concept and logic of *Dr. Wise Owl's Quiz Game*. I conceived and pulled all this together today. Development in App Inventor—and I'm sure we all agree on that now—is fast and lets us do lots with very few blocks.

Now, we take a quick look at the **GameClient** component.

GameClient component

Well, it's a games chapter so I feel compelled to mention the **GameClient** component—part of its blocks are shown in **Blocks Editor** next. This component is meant for use in communicating with online game servers, such as those for popular role-playing games.

Alas, **GameClient** is still in the **Not ready for prime time** and, judging by all the posts I've read, a little less ready than some of the other components in there. By the way, it runs using Python, which you'll need to set up on a server. So, consider it mentioned, but keep your eye on it—those great AI developers keep on giving us more and more great stuff!



Summary

In this chapter, we played with animation techniques using balls and image sprites. We learned how to build animated splash screens with music and all, and played some *SpaceBall*.

Bash the Alien! we found to be an example of arcade-style games requiring fast reaction times as the little alien did his best to make us look silly.

Then, we did a major example of a knowledge game with *Dr. Wise Owl's Quiz Game*. This app shows how fast we can develop useful apps using App Inventor.

And, we discovered that the **GameClient** component is still in training.

That completes this book except for a few pages of extra and useful stuff in the appendices.

Hope you enjoyed this voyage through App Inventor as much as I did. AI is fast-changing, and I'm sure we'll have yet more features to play with even before this book reaches print, but I think I've given you the tools to understand and use them as they come.

Thank you!

Thanks for riding along on the Google App Inventor express. We've explored and accumulated a ton of non-programming, block-pushing techniques (the easy and fun way to build apps). Now, let your creativity take charge as you turn out all those great apps!

I've written over 100 books now, but it's still a thrill to actually *finish* one.

But, there's no rest for me—today, literally as I turn in this completed book, I am working on my next book for Packt Publishing, one about Google Plus.

So, onward and upward and, later on, I hope to be back doing an advanced book on App Inventor. I truly love it!

Keep track of me at my personal blog, `http://ralphroberts.net`, and if you have any comments, feel free to e-mail me at `ralph@abooks.com`.

Bless you all, my friends!

A
Links and Resources

I've collected some extras for you guys as a parting gift in this Appendix.

Websites

Useful learning and other resource links are here—please make a note that the `googlelabs` URLs will be changing as Google phases it out. But, App Inventor continues full speed, and its new address will be widely posted on the Internet and no doubt on the sites below before the change occurs. The following are the App Inventor URL we will use: `http://appinventorbeta.com/learn`—the main site for tutorials and links about AI.

`http://appinventorbeta.com/forums`—the official AI forums; very helpful group of folks including, I hope, me. Read it daily for all the latest news and tips about AI. Great community!

`http://amerkashi.wordpress.com/`—the home of *AppToMarket*, which is one of the two best ways to place your app on Android Market, and the other good stuff from Hossein Amerkashi.

`http://www.taiic.com/downloads/`—*Marketizer* by Gene Kupfer, the other way of uploading to Market, is here, and also the various other goodies, which bring us to...

`http://www.tair.info/`—the App Inventor Repository, thanks to hard work by Anthony Barnes, Gene Kupfer, and others, has tons of tutorials and downloads. I find it exceptionally useful.

`http://www.bigdaddyapp.com/`—Roger Belk (Big Daddy) has an interesting site I enjoy. Roger (and the people mentioned by name here) are people I consider to be true heroes of AI, always helpful and delightful.

`http://ai.kittywolf.net/index.php/Main_Page`—I consider Shival Wolf from down in the Land of Oz to be the expert in interfacing AI with PHP, MySQL, and other online services. An extensive site.

`http://blocks123.net`—another extensive site with all sorts of tutorials by Ed Joybells.

`http://www.appinventor.org/`—Professor David Wolber's site, a pioneer in teaching App Inventor, author, expert, and a great guy.

`http://android.jwtyler.com/`—J.W. (Jason) Tyler is another AI expert with some very worthwhile and helpful video tutorials on his site.

Search `http://YouTube.com` for *app inventor*. There are endless pages of interesting tutorials and other videos about AI.

And, as ever, remember Google Search is your best friend in finding stuff!

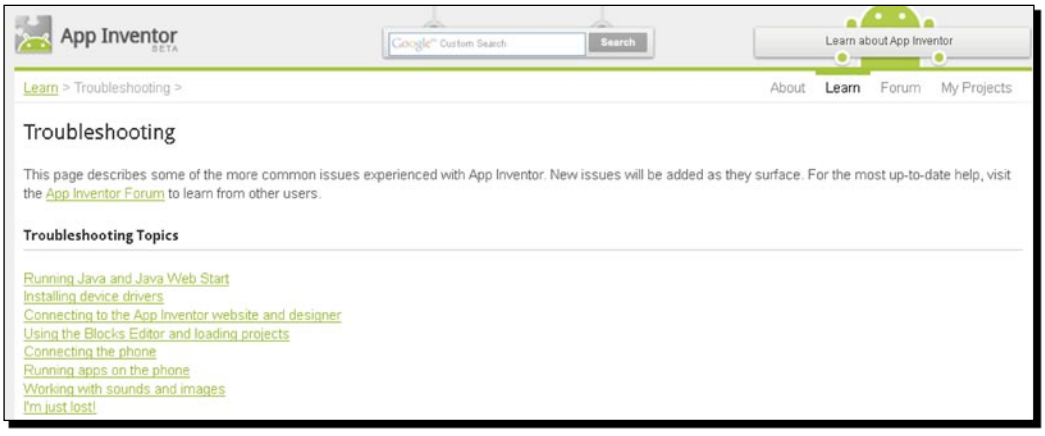
Troubleshooting

Troubleshooting is basically a process of elimination. You start disconnecting parts until you find the troublemaker. Fix and repair.

In App Inventor, there are two types of problems. The first is related to problems in the part of AI that runs on your local computer—such as Java problems, incorrect installation, the Android SDK (Software Development Kit) not running, the right drivers for your test device not in place, and so forth.

For solutions to these, first check `http://appinventorbeta.com/learn/troubleshooting.html`, shown in the following screenshot (that's the current URL, it will soon change due to Google sunsetting GoogleLabs, but keep checking the App Inventor forums site and you'll know when it changes).

Also, the App Inventor forums themselves have lots of helpful experts, but do a Google search first—chances are 14,000 other people had the same problem and solutions are posted on the web.


App Inventor
BETA
Search
Learn about App Inventor
Learn > Troubleshooting >
About
Learn
Forum
My Projects
Troubleshooting
This page describes some of the more common issues experienced with App Inventor. New issues will be added as they surface. For the most up-to-date help, visit the App Inventor Forum to learn from other users.
Troubleshooting Topics
Running Java and Java Web Start
Installing device drivers
Connecting to the App Inventor website and designer
Using the Blocks Editor and loading projects
Connecting the phone
Running apps on the phone
Working with sounds and images
I'm just lost!

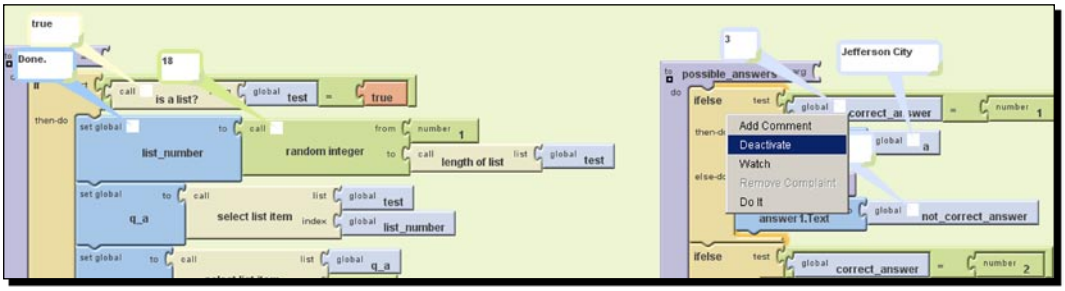
In an app that is not working right, AI gives us several useful troubleshooting tools. For example, in *Mr. Wise Owl's Quiz Game* (example in the last chapter) it gave me trouble at first—it kept crashing. Not good; people sort of expect examples in books to work, eh?

I used the AI built-in troubleshooting tools to (after a bit of time) find the problem.

To use these tools yourself, right-click on blocks, and a menu comes up with useful options for troubleshooting:

1. **Deactivate**—stops the component and any blocks attached to it from working. If the app no longer crashes, you have a problem in those blocks.
2. **Watch**—brings up a little cartoon-like balloon showing the content of the block during operation of the app. Right-click the block again to stop watching.
3. **Do It**—runs only that block or block group.

Here's what troubleshooting my quiz app looked like (now everything's working and the right content shows):



What was my problem? I had the app accessing a list before the list was populated. That meant it was encountering an empty string, which was not to its taste, hence the crash (meaning the app stopped working and my test device was no longer connected to the **Blocks Editor**).

The following explains how you fix that problem. Put in an **if** block and test to make sure the global variable really has a list in it before committing to whatever operation this block group does. Yes, we call this **error trapping**; that is, catching errors before they disrupt the app.

Download from Wow! eBook <www.wowebook.com>

Using the Android SDK

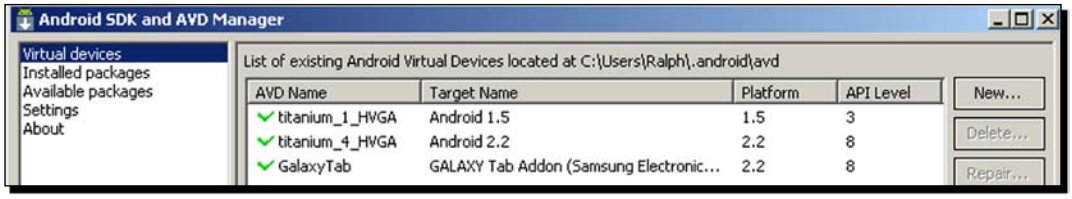
The Android SDK or Software Development Kit is run by the Java computer language underneath App Inventor. It's what makes things like our neat app creation by simply pulling together blocks and the use of an emulator as a test device possible.

Normally, we could care less in the same way we drive a car without thinking about what position the pistons are in and if the spark plugs spark. Still, the SDK is there for our use.

On Windows computers, the SDK should be in the `c:\program files\android` directory if you installed it as suggested in *Chapter 1*, *Obtaining and Installing Google App Inventor*. Clicking on the `SDK Manager.exe` opens the main SDK program.

Using this program, you can do things such as install additional emulators for use in **Blocks Editor**. For example, in mine I've added an emulator for the Galaxy tablet computer as shown in the next screenshot (see *Chapter 1*, *Obtaining and Installing Google App Inventor* again for more details).

Oh, and also check `SDK Manager.exe` occasionally to make sure all updates have been applied.



If you actually want to program in Java directly, you'll want to install Eclipse, an open source IDE (Integrated Development Environment) that makes it much easier to work with Java and the SDK.

Of perhaps more immediate use to you is the `ddms.bat` batch file (found in the `tools` directory), which opens the Dalvik Debug Monitor as we saw earlier in the book. It does several nice things including screen captures of your test device as shown in the following screenshot, and it is indispensible if you are creating art for the apps you post on Market.

Java Bridge

Finally, something that's going to be hot you may want to look into. The Java Bridge allows programming of the SDK in Eclipse while including App Inventor components! The best of both worlds—access to the lower levels of your phone than AI provides (such as using the hardware buttons) while still having the ease of doing things by merely pulling blocks together.

Some good tutorials on Java Bridge are now available in the App Inventor forums. Check the App Inventor for Education forum for discussions on Java Bridge.

A good source for tutorials on Java Bridge is at:
`http://code.google.com/p/apptomarket/downloads/list`

B
Last-Minute Update

This book was—so I thought—completed on August 3, 2011. Naturally, on August 5, a new update was released. To make this book as absolutely complete as possible, here it is:

Features:

- Allow **Player** component to work with streaming URLs on Android 1.6 or greater
- Added **GetPixelColor**, **GetBackgroundPixelColor**, and **SetBackgroundPixelColor** to **Canvas**. (issue #485)
- Added make color and split color blocks to the **Built-In**/**Color** drawer

Bug Fixes:

- Fixed issue #1750 - **Web.HtmlTextDecode** caused runtime error on phones older than Gingerbread
- Fixed issue #1756 - App crashes when phone turned to landscape
- Fixed bug where **Canvas.save** fails on a very large canvas

Of the previous changes, the one I find most exciting is the new **make color** block in the **Built-In**/**Color** drawer. Instead of those long negatives numbers we've used to set colors throughout the book, it's now possible to use standard RGB (Red, Green, Blue) designations (0-255) for each of the three.

In addition to this standard color nomenclature, an alpha channel was also added for controlling transparency. It is now possible, for example, to (using delays) fade labels in and out. Transitions! Something I've wanted a lot. I am really looking forward to playing with that.

Status of App Inventor

Finally—also on August 3, just as I finished this book, Google announced it was discontinuing support of App Inventor (part of its general discontinuation of Google Labs) and releasing it to open source. You can imagine my initial reaction after all this work.

However, as seen previously, support has continued with the release of another upgrade, and App Inventor continues to progress.

So, the news is good, and I look forward to clicking blocks and creating apps for years to come.

I hope this book helps you as much as it has me in mastering App Inventor.

— Ralph Roberts, August 6, 2011

Last-minute update

Good news on the future of App Inventor!

On August 16, 2011, Hal Abelson, Professor of Computer Science and Engineering, MIT (and one of the creators of App Inventor), posted an announcement about AI's future on the Google Research blog. In part it reads:

> *"MIT and Google have a long-standing relationship based on mutual interests in education and technology. Today, we took another step forward in our shared goals with the establishment of the MIT Center for Mobile Learning, which will strive to transform learning and education through innovation in mobile computing. The new center will be actively engaged in studying and extending App Inventor for Android, which Google recently announced it will be open sourcing.*
>
> *"The new center, housed at MIT's Media Lab, will focus on designing and studying new mobile technologies that enable people to learn anywhere, anytime, with anyone. The center was made possible in part by support from Google University Relations and will be run by myself and two distinguished MIT colleagues: Professors Eric Klopfer (science education) and Mitchel Resnick (media arts and sciences)."*

This means App Inventor—with support from both MIT (Massachusetts Institute of Technology) and from Google—continues development and becomes even more of an open source project. The roots of AI are at MIT as well as another graphic programming language, Scratch.

It's all an exciting synthesis promising continued growth for App Inventor, its wonderful community of already over 100,000 users (according to one of the creators of AI, Hal Abelson), and the added boost of MIT's technological incubation.

I expect great things.

And, you can bet I plan on creating apps with App Inventor for years to come.

There's simply nothing else so easy to master, yet so powerful. Playing with blocks? Wonderful!

— Ralph Roberts, August 25, 2011

C
Final Last-Minute Update

In closing, the following e-mail was just received from the Google App Inventor Team and also widely published on the Internet. These great men and women have really worked hard in developing App Inventor and also to keep it alive when Google decided to close out so many of its projects recently.

This is very good news for all of us app inventors, and we reproduce the letter here for you:

Letter from the App Inventor Team

Dear App Inventor User,

As a result of the recent changes to Google Labs and App Inventor, effective immediately, the URL for App Inventor will change from `appinventor.googlelabs.com` to `appinventorbeta.com`. This URL change will not have an impact on your projects stored in App Inventor. All data that you see in your `appinventor.googlelabs.com` account, as well as documentation and e-mail forums, will be available at `appinventorbeta.com`.

As we announced on the App Inventor Announcement Forum, Google will end support for App Inventor and open source the code base at the end of this year. Additionally, in order to ensure the future success of App Inventor, Google has funded the establishment of a Center for Mobile Learning at the MIT Media Lab, where MIT will be actively engaged in studying and extending App Inventor. This transition will happen at the end of 2011. At that time, you will need to download your data from `appinventorbeta.com` in order to continue working with it in the open source instance of App Inventor. In the coming months, we will send you detailed instructions on how to download your data. Please visit the App Inventor user forums to get future updates on App Inventor.

The App Inventor Team

D

Pop Quiz Answers

Chapter 1

Obtaining and Installing Google App Inventor

1	b
2	c
3	d

Chapter 3

Playing with Blocks

1	c
2	d
3	c

Chapter 5

Apps That Communicate

1	c
2	b
3	d
4	d

Chapter 7

Apps That Surf the Web

1	d
2	c
3	d

Index

Download from Wow! eBook <www.wowebook.com>

Symbols

$_REQUEST 252
.apk file extension 47
.zip file 41

A

AccelerometerSensor component
about 257, 263
acceleration, viewing on phone 264, 265
gravity, viewing on phone 264, 265
ActivityStarter
about 138, 139
bookmark apps, building 139, 140
ActivityStarter parameter 261
Add_Friend.click blocks group 176
advanced tab
in block editor 254, 255
altitude
finding 272
android.action.intent.VIEW 184
Android phones 167
Android SDK
about 80, 318, 320
using 320
animation
about 283, 284
ball, bouncing 284, 285
blocks, playing with 290-294
features, adding to game 294, 295
things, colliding 289, 290
animation components
about 70
Ball 71
ImageSprite 71
app
building 275, 276
current address, getting 276, 277
end, building 275, 276
GPS, turning off 280
GPS, turning on 280
home location, saving 277, 278
latitude, getting 276, 277
longitude, getting 276, 277
multiple websites, displaying 250, 252
URL, for downloading 273
used, for texting friends 182, 183
App Inventor
about 8, 327
apps, designing 50, 51
Blocks Editor 51, 78
call, making with 168, 169
changelog 134, 249
Colors block 129, 130
concepts 133
Control blocks 127
CSV list blocks 120, 122
Definition blocks 85
Designer 51
devices, configuring 29-32
drivers, configuring on Linux 26-28
drivers, configuring on Mac OS X 26
drivers, configuring on Windows 28, 29

email, sending from 184-193
Forum section 12
individual component blocks 131
installing, locally 15
installing, on GNU/Linux 16-18
installing, on Mac OS X 16
installing, on Windows 18, 19
issues 318
key concept 92
last-minute update 324
Lists blocks 112-120
logging into 9-13
Logic blocks 126
Math blocks 122-125
My Projects section 13
operating system requisites 13
other changes 256
parts 19, 51
phone, connecting to 48-50
resource links 317, 318
status 324
test device 20, 51
Text blocks 104-109
troubleshooting issues 318, 319
troubleshooting tools 319
URL, for official AI forums 317
URL, for tutorials 168, 317
URLs 317, 318
web interface, using 36, 37
App Inventor, installing
on GNU/Linux 16-18
on Mac OS X 16
on Windows 18, 19
App Inventor account
getting 8, 9
App Inventor animation
basics 295-297
App Inventor Announcement Forum 327
App Inventor Blocks Editor 9, 19, 21, 51
App Inventor Designer 9, 10, 19, 36, 51
App Inventor in Action video 11
App Inventor Repository 41, 317
App Inventor Team
about 327
letter 327
App Inventor Web Interface
Design 44-48
My Projects 37-44
using 36, 37
apps
designing 50, 51
publishing 243-246
ArrSoft 171
AVLnews, example 225
Azimuth 269

B

Ball, animation component 284
Ball1.Bounce block 286
Ball1.EdgeReached block 286
Ball component 71
BarcodeScanner 143
Bash the Alien!
about 303, 304
block, throwing to 305-307
basic components
about 35, 51, 52
Button 52-57
Canvas 58, 59
Checkbox 59
Clock 60
Image 60
Label 60, 61
ListPicker 61
PasswordTextBox 62
TextBox 62
TinyDB 63
block groups
collapsing 82, 83
expanding 82, 83
blocks 77
Blocks Editor
about 9, 78, 184, 320
advanced tab 254, 255
top bar 78, 79
Bluetooth 75, 144-148
bouncing ball
following 58, 59
button
adding 53-57
configuring 53-57
Button component
about 52

button, adding 53-57
button, configuring 53-57
buttons
vibrating 69, 70

C

calculate procedure 279
call
making, with App inventor 168, 169
CallFriends app
about 168, 171
building 171, 172
modifying 177, 178
non-visible components 173
source, retrieving 172-177
Camera component
about 65
photo, shooting 65, 66
Canvas component 251
about 58
bouncing ball, following 58, 59
Center for Mobile Learning 327
changeable lists
about 202
friends list apllication, completing 206
input screen for friends list, building 202-205
changelog
about 134, 249
modifications 256
Checkbox component 59
circle area
calculating, function Design procedure used 88-91
calculating, function procedure used 92, 93
Clock1.TimerInterval block 301
Clock component 60
Color Chart page 130
Colors blocks 129, 130
compass
building 270, 271
components 35
component types, App Inventor
animation 70
basic 51, 52
LEGO® MINDSTORMS® 75
media 64
screen arrangement 74
sensor 74
social 72
concepts
about 135
component size, specifying 136, 137
debugging 135
Do It 136
images, accessing 137
live development 135
Remove Complaint 135
sounds, accessing 137
testing 135
Watch 135
ContactPicker component 72
contains block 108
Control blocks
about 127
loop, creating 128, 129
CSV (comma-separated values) 228
CSV list blocks 120, 122
CSV Table
converting 209-211

D

Dalvik Debug Monitor 80, 320
data 200
database 200
database records
CSV Table, converting 209-211
handling 207, 208
datum 200
ddms.bat batch file 320
Definition blocks
about 85
dummy block 103, 104
name block 101-103
procedure block 98, 99
procedureWithResult block 86, 87
variable block 100
Design, App Inventor Web Interface
about 44-48
phone, connecting to App Inventor 48-50
devices
configuring 29-32
distance procedure 279

division 124
downcase block 107
downloading
drivers 25
projects 40-42
drivers
about 25
configuring, on Linux 26-28
configuring, on Mac OS X 26
configuring, on Windows 28, 29
downloading 25
issues 26
searching 25
Droid
connecting, to Ubuntu 26-28
Droid 2 25, 168
dsrcid variable 230
dummy block 103, 104

E

eBay link app
building 222-224
email
sending, from App Inventor 184-193
EmailPicker component 72
emulator
opening up 20-25
running 20-25
setting up 20
emulators 79
error trap 97, 98
error trapping 319
Exit button 266, 309

F

Facebook 193, 196, 197
friends list application, changeable lists
building 206
function
about 87
example 87
function Design procedure
used, for calculating area of circle 88-91
function procedure
used, for calculating area of circle 92, 93
Fusion Tables 227-235
FusiontablesControl 163, 164

G

Galactic Headlines 161
GameClient 165
GameClient component 314
get method, Web component 213, 217, 219
Get Mileage Log button 234
Global Positioning Satellites (GPS) 273
GNU/Linux
App Inventor, installing on 16-18
Google Account
signing up for 8, 9
Google account 9
Google App Inventor. *See* **App Inventor**
Google Fusion Tables 75
Google Labs 327
Google Maps
about 258
location, finding by GPS 259, 260
location, finding by network 258
Google Mobilizer! 226
Googlenauts 37
Google Tables 229
GoogletablesControl component 229
GPS
about 75
turning off 280
turning on 280

H

HandcentSMS action 142

I

I Love You app 186-192
Image component 60
ImagePicker component 67
ImageSpite component 71
ImageSprite, animation component 284
ImageSprite component 287
individual component blocks 131
input screen, changeable lists
for friends list, building 202-205

installing
App Inventor, on GNU/Linux 16-18
App Inventor, on Mac OS X 16
App Inventor, on Windows 18, 19
Java 15

J

Java
about 14
installing 15
Java Bridge
about 321
URL, for tutorials 321
join block 105
json_encode command 242

K

knowledge games 307-314

L

Label component 60, 61
LAMP 161
LEGO® MINDSTORMS® components 75
Lego Mindstorms NXT 75
length block 105
Linux
drivers, configuring on 26-28
List blocks
about 113
examples 113-120
ListPicker component 61
lists
about 112, 200
changeable lists 202
displaying 110-112
static lists 200-202
LocationSensor 257, 273
location sensor
AccelerometerSensor 257
Global Positioning Satellites (GPS) 257
LocationSensor 257
network method 257
OrientationSensor 257
using 257, 258
Logic blocks 126
Logic drawer 126
loop
creating 128, 129

M

Mac OS X
App Inventor, installing on 16
drivers, configuring on 26
make text block 105
Marketizer 143
Math blocks 122-125
media components
about 35, 64
Camera 65, 66
ImagePicker 67
Player 67-69
Sound 69, 70
VideoPlayer 70
media files, accessing ways
application assets 137
phones sd card 137
web 137
music player
building 68, 69
My Projects, App Inventor Web Interface
about 37
new project, creating 38, 40
projects, downloading 40-42
source code, uploading 42-44

N

name block 101-103
New emulator button 81
Notifier
about 148-151, 274
with two buttons 152-154
Not Ready for Prime Time drawer
about 163
FusiontablesControl 163, 164
GameClient 165
SoundRecorder 165
voting 165

O

Open the Blocks Editor button 45
operating system requisites, for App Inventor 13
OrientationSensor 257
Other Stuff Drawer
about 138
ActivityStarter 138, 139
BarcodeScanner 143
Bluetooth 144-148
Notifier 148-151
SpeechRecognizer 154, 155
TextToSpeech 157, 158
TinyWebDB 159, 160
Web 161-163

P

Package for Phone button 46
parsing, Web component
about 213, 214
used, for breaking states 214-216
PasswordTextBox component 62
personal account
creating, on Google App Inventor site 8
phone
connecting, to App Inventor 48-50
screens, capturing from 80-82
phone app
building 168, 169
PhoneCall component 73, 168
PhoneNumberPicker component 73, 168
photo
shooting 65, 66
pi are square app
finishing 94-96
PicCall 168
Pitch 269
Player component
about 67
music player, building 68, 69
post, Web component 213
post method
using, with web component 252, 253
procedure block 98, 99
procedureWithResult block 86
project
creating 38, 40
downloading 40-42

R

randomizing 124
Redo button 79
replace all block 111, 112
Reset button 309
Roll 269
rounding 124

S

Saved button 79
Screen1.Initialize block 301
screen arrangement components 74
screens
capturing, from phone 80-82
segment block 111
sensor components 74
Short Messaging Service. *See* **SMS**
Show Barcode option 46
Six Screen Template 169, 170
SMS 178
social communication
about 193
Facebook 196, 197
methods, for searching Twitter 193-196
other social media sites 196, 197
social components
about 35, 72
ContactPicker 72
EmailPicker 72
PhoneCall 73
PhoneNumberPicker 73
Texting 73
Twitter 74
sockets
about 59
Sound component
about 69
buttons, vibrating 69, 70
SoundRecorder 165
source code
uploading 42-44

SpeechRecognizer
about 154
date stamped voice note taker, building 154-156
time stamped voice note taker, building 154-156
splash screens
about 297
designing 297-300
splash, making with splash page 300-303
split at any block 111
split at first block 108
split at first of any block 109
split at spaces block 111
split block 109
starts at block 108
static lists 200-202
subroutine 87
sudo privileges 17, 26

T

tablet computers 167
tAIR. *See* **App Inventor Repository**
test> block 106
test device 51
text< block 106
text= block 106
text block 104
Text Blocks
about 104
contains 108
downcase 107
join 105
length 105
list, displaying 110-112
make text 105
replace all 111
segment 111
split 109
split at any 111
split at first 108
split at first of any 109
split at spaces 111
starts at 108
test> 106
text 104
text< 106
text= 106
trim 107
upcase 107
TextBox component 62
TextFriends app 182, 183
texting 178
texting app
building 178-181
Texting component 73
texts
sending 178
TextToSpeech
reading aloud 157, 158
Text to Speech 75
TinyDB 211, 212, 274
TinyDB component
about 63, 168
working 63
TinyWebDB
about 75, 159, 228
persistent data, storing on web 159, 160
tools directory 320
top bar, Blocks Editor
about 78, 79
Redo button 79
Saved button 79
Undo button 79
trigonometric functions 125
trim block 107
troubleshooting 318
tweets 193
Twitter
about 74, 193
methods, for searching 193-196

U

Ubuntu
Droid, connecting to 26-28
udev rules directory 27
Undo button 79
unwanted blocks
deleting 84
upcase block 107

V

variable block 100
VideoPlayer component 70
voting 165

W

Web 161-163, 228
Web component
about 212
get method 213
parsing 213
post 213
post method, using with 252, 253
web pages
bringing, into apps 250
multiple websites, displaying inside app 250, 252
webresults.php 241
websites
browsing 221
browsing, adding to apps 222
WebViewer type component 250
Whac-a-Mole™ 303
what you see is what you get. *See* **WYSIWYG**
Windows
App Inventor, installing on 18, 19
drivers, configuring on 28, 29
World Wide Web (WWW) 222
WYSIWYG 35, 169

Thank you for buying
Google App Inventor Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.

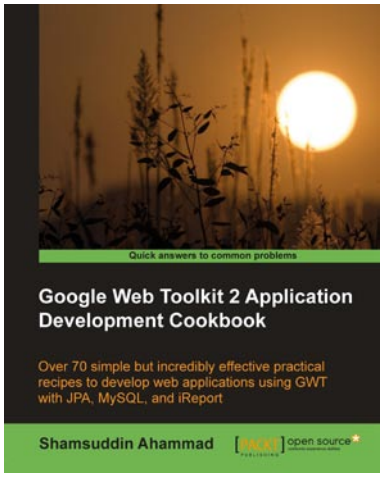
Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Download from Wow! eBook <www.wowebook.com>

[PACKT]
PUBLISHING


Quick answers to common problems
Google Web Toolkit 2 Application Development Cookbook
Over 70 simple but incredibly effective practical recipes to develop web applications using GWT with JPA, MySQL, and iReport
Shamsuddin Ahammad
open source

Google Web Toolkit 2 Application Development Cookbook

ISBN: 978-1-84951-200-8 Paperback: 244 pages

Google Web Toolkit book and eBook - over 70 simple but incredibly effective practical recipes to develop web applications using GWT with JPA, MySQL and iReport

1. Create impressive, complex browser-based web applications with GWT 2
2. Learn the most effective ways to create reports with parameters, variables, and subreports using iReport
3. Create Swing-like web-based GUIs using the Ext GWT class library
4. Develop applications using browser quirks, Javascript, HTML scriplets from scratch

---

Community Experience Distilled
Google App Engine Java and GWT Application Development
Build powerful, scalable, and interactive web applications in the cloud
Daniel Guermeur
Amy Unruh
PACKT

Google App Engine Java and GWT Application Development

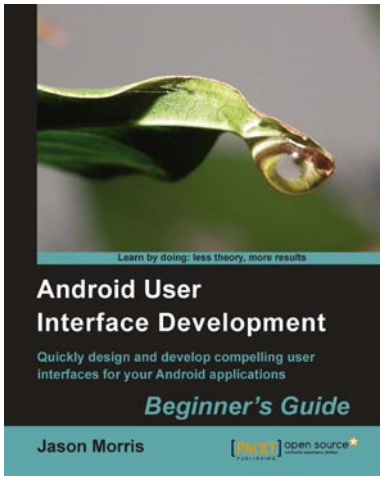
ISBN: 978-1-84969-044-7 Paperback: 480 pages

Google App Engine Java and GWT Application Development book and eBook - Build powerful, scalable, and interactive web applications in the cloud

1. Comprehensive coverage of building scalable, modular, and maintainable applications with GWT and GAE using Java
2. Leverage the Google App Engine services and enhance your app functionality and performance
3. Integrate your application with Google Accounts, Facebook, and Twitter

Please check **www.PacktPub.com** for information on our titles

[PACKT]
PUBLISHING

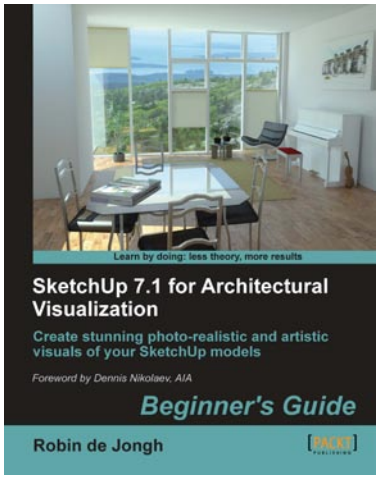

Learn by doing: less theory, more results
Android User Interface Development
Quickly design and develop compelling user interfaces for your Android applications
Beginner's Guide
Jason Morris
[PACKT] open source

Android User Interface Development: Beginner's Guide

ISBN: 978-1-84951-448-4 Paperback: 304 pages

Quickly design and develop compelling user interfaces for your Android applications

1. Leverage the Android platform's flexibility and power to design impactful user-interfaces
2. Build compelling, user-friendly applications that will look great on any Android device
3. Make your application stand out from the rest with styles and themes
4. A practical Beginner's Guide to take you step-by-step through the process of developing user interfaces to get your applications noticed!


Learn by doing: less theory, more results
SketchUp 7.1 for Architectural Visualization
Create stunning photo-realistic and artistic visuals of your SketchUp models
Foreword by Dennis Nikolaev, AIA
Beginner's Guide
Robin de Jongh
[PACKT]

SketchUp 7.1 for Architectural Visualization: Beginner's Guide

ISBN: 978-1-847199-46-1 Paperback: 408 pages

Create stunning photo-realistic and artistic visuals for your Google SketchUp models

1. Create picture-perfect photo-realistic 3D architectural renders for your SketchUp models
2. Post-process SketchUp output to create digital watercolor and pencil art
3. Follow a professional visualization studio workflow
4. Make the most out of SketchUp with the best free plugins and add-on software to enhance your models